

AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
37	AAA	Z0	Invalid	Valid	ASCII adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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
        AX := AX + 106H;
        AF := 1;
        CF := 1;
      ELSE
        AF := 0;
        CF := 0;
    FI;
    AL := AL AND 0FH;
  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	Z0	Invalid	Valid	ASCII adjust AX before division.
D5 ib	AAD imm8	Z0	Invalid	Valid	Adjust AX before division to number base imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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	Z0	Invalid	Valid	ASCII adjust AX after multiply.
D4 ib	AAM imm8	Z0	Invalid	Valid	Adjust AX after multiply to number base imm8.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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	Z0	Invalid	Valid	ASCII adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 0FH) > 9) or (AF = 1)
      THEN
        AX := AX - 6;
        AH := AH - 1;
        AF := 1;
        CF := 1;
        AL := AL AND 0FH;
      ELSE
        CF := 0;
        AF := 0;
        AL := AL AND 0FH;
    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 ib	ADC AL, imm8	I	Valid	Valid	Add with carry imm8 to AL.
15 iw	ADC AX, imm16	I	Valid	Valid	Add with carry imm16 to AX.
15 id	ADC EAX, imm32	I	Valid	Valid	Add with carry imm32 to EAX.
REX.W + 15 id	ADC RAX, imm32	I	Valid	N.E.	Add with carry imm32 sign extended to 64-bits to RAX.
80 /2 ib	ADC r/m8, imm8	MI	Valid	Valid	Add with carry imm8 to r/m8.
REX + 80 /2 ib	ADC r/m8, imm8	MI	Valid	N.E.	Add with carry imm8 to r/m8.
81 /2 iw	ADC r/m16, imm16	MI	Valid	Valid	Add with carry imm16 to r/m16.
81 /2 id	ADC r/m32, imm32	MI	Valid	Valid	Add with CF imm32 to r/m32.
REX.W + 81 /2 id	ADC r/m64, imm32	MI	Valid	N.E.	Add with CF imm32 sign extended to 64-bits to r/m64.
83 /2 ib	ADC r/m16, imm8	MI	Valid	Valid	Add with CF sign-extended imm8 to r/m16.
83 /2 id	ADC r/m32, imm8	MI	Valid	Valid	Add with CF sign-extended imm8 into r/m32.
REX.W + 83 /2 id	ADC r/m64, imm8	MI	Valid	N.E.	Add with CF sign-extended imm8 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)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

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

DEST := DEST + SRC + 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.

ADCX—Unsigned Integer Addition of Two Operands With Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
66 REX.w 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

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

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is 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, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

Operation

IF OperandSize is 64-bit

```
THEN CF:DEST[63:0] := DEST[63:0] + SRC[63:0] + CF;
ELSE CF:DEST[31:0] := DEST[31:0] + SRC[31:0] + CF;
```

FI;

Flags Affected

CF is updated based on result. OF, SF, ZF, AF, and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

```
unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#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.
#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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

ADD—Add

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 id	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 ib	ADD r/m8, imm8	MI	Valid	Valid	Add imm8 to r/m8.
REX + 80 /0 ib	ADD r/m8 [*] , imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m8.
81 /0 iw	ADD r/m16, imm16	MI	Valid	Valid	Add imm16 to r/m16.
81 /0 id	ADD r/m32, imm32	MI	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 id	ADD r/m64, imm32	MI	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 ib	ADD r/m16, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 ib	ADD r/m32, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 ib	ADD r/m64, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX + 00 /r	ADD r/m8 [*] , r8 [*]	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX + 02 /r	ADD r8 [*] , r/m8 [*]	RM	Valid	N.E.	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add 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)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

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 prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST := DEST + SRC;

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 Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Add packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Add packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Add packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds two, four or eight packed double precision floating-point values from the first source operand to the second source operand, and stores the packed double precision floating-point result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a 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 (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Vector Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] := SRC1[i+63:i] + SRC2[63:0]

ELSE

DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VADDPD (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]

DEST[MAXVL-1:256] := 0

.

VADDPD (VEX.128 Encoded Version)

DEST[63:0] := SRC1[63:0] + SRC2[63:0]

DEST[127:64] := SRC1[127:64] + SRC2[127:64]

DEST[MAXVL-1:128] := 0

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[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPD __m512d __mm512_add_pd (__m512d a, __m512d b);

VADDPD __m512d __mm512_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);

VADDPD __m512d __mm512_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);

VADDPD __m256d __mm256_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);

VADDPD __m256d __mm256_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);

VADDPD __m128d __mm_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);

VADDPD __m128d __mm_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);

VADDPD __m512d __mm512_add_round_pd (__m512d a, __m512d b, int);

VADDPD __m512d __mm512_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);

VADDPD __m512d __mm512_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);

ADDPD __m256d __mm256_add_pd (__m256d a, __m256d b);

ADDPD __m128d __mm_add_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

ADDPS—Add Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 58 /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.128.0F.WIG 58 /r VADDPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed single precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.128.0F.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Add packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.0F.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Add packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F	Add packed single precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds four, eight or sixteen packed single precision floating-point values from the first source operand with the second source operand, and stores the packed single precision floating-point result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a 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 (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation**VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0

VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]

ELSE

DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0

VADDPS (VEX.256 Encoded Version)

$\text{DEST}[31:0] := \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] := \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] := \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] := \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[159:128] := \text{SRC1}[159:128] + \text{SRC2}[159:128]$
 $\text{DEST}[191:160] := \text{SRC1}[191:160] + \text{SRC2}[191:160]$
 $\text{DEST}[223:192] := \text{SRC1}[223:192] + \text{SRC2}[223:192]$
 $\text{DEST}[255:224] := \text{SRC1}[255:224] + \text{SRC2}[255:224]$
 $\text{DEST}[\text{MAXVL}-1:256] := 0$

VADDPS (VEX.128 Encoded Version)

$\text{DEST}[31:0] := \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] := \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] := \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] := \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[\text{MAXVL}-1:128] := 0$

ADDPS (128-bit Legacy SSE Version)

$\text{DEST}[31:0] := \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] := \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] := \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] := \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[\text{MAXVL}-1:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`VADDPS __m512 __mm512_add_ps (__m512 a, __m512 b);`
`VADDPS __m512 __mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);`
`VADDPS __m512 __mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);`
`VADDPS __m256 __mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);`
`VADDPS __m256 __mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);`
`VADDPS __m128 __mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);`
`VADDPS __m128 __mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);`
`VADDPS __m512 __mm512_add_round_ps (__m512 a, __m512 b, int);`
`VADDPS __m512 __mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);`
`VADDPS __m512 __mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);`
`ADDPS __m256 __mm256_add_ps (__m256 a, __m256 b);`
`ADDPS __m128 __mm_add_ps (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

ADDSD—Add Scalar Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5B /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 5B /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F2.0F.W1 5B /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Add the low double precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds the low double precision floating-point values from the second source operand and the first 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 operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSD (EVEX Encoded Version)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] := SRC1[63:0] + SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                               ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

VADDSD (VEX.128 Encoded Version)

```

DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

ADDSD (128-bit Legacy SSE Version)

```

DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VADDSD __m128d __mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d __mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d __mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d __mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d __mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d __mm_add_sd (__m128d a, __m128d b);

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”

ADDSS—Add Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 58 /r VADDSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Add the low single precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LLIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Add the low single precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Adds the low single precision floating-point values from the second source operand and the first 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 operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:32) of the corresponding the destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSS (EVEX Encoded Versions)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] := SRC1[31:0] + SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                               ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

VADDSS DEST, SRC1, SRC2 (VEX.128 Encoded Version)

```

DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

ADDSS DEST, SRC (128-bit Legacy SSE Version)

```

DEST[31:0] := DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VADDSS __m128 __mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 __mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 __mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 __mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 __mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 __mm_add_ss (__m128 a, __m128 b);

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”

ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract

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

Instruction Operand Encoding

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

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-point 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 (MAXVL-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 (MAXVL-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.

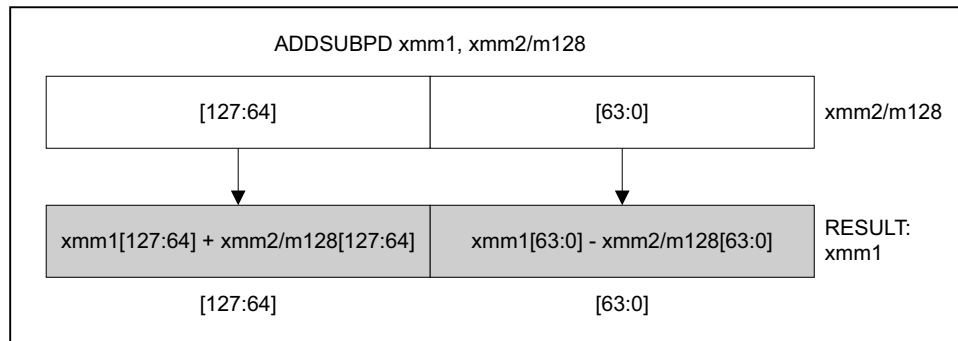


Figure 3-3. ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract

Operation

ADDSUBPD (128-bit Legacy SSE Version)

DEST[63:0] := DEST[63:0] - SRC[63:0]
 DEST[127:64] := DEST[127:64] + SRC[127:64]
 DEST[MAXVL-1:128] (Unmodified)

VADDSUBPD (VEX.128 Encoded Version)

DEST[63:0] := SRC1[63:0] - SRC2[63:0]
 DEST[127:64] := SRC1[127:64] + SRC2[127:64]
 DEST[MAXVL-1:128] := 0

VADDSUBPD (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

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 Table 2-19, "Type 2 Class Exception Conditions."

ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract

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

Instruction Operand Encoding

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

Description

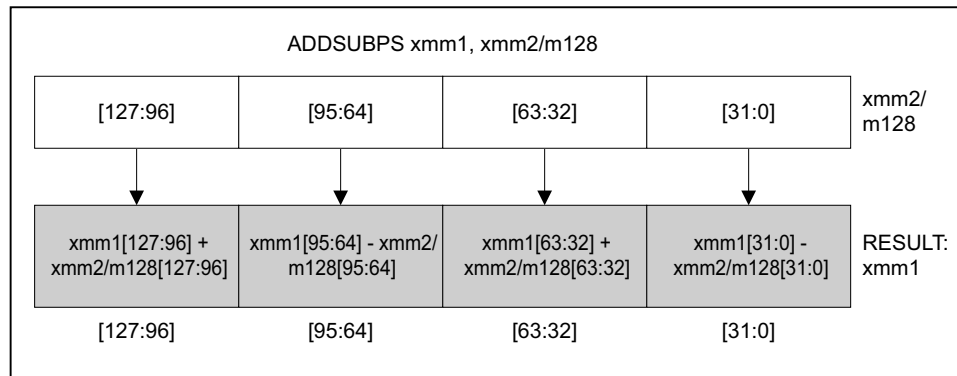
Adds odd-numbered single precision floating-point values of the first source operand (second operand) with the corresponding single precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). 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 (MAXVL-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 (MAXVL-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 Precision Floating-Point 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[MAXVL-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[MAXVL-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 Table 2-19, “Type 2 Class Exception Conditions.”

ADOX — Unsigned Integer Addition of Two Operands With Overflow Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F6 /r ADOX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with OF, r/m32 to r32, writes OF.
F3 REX.w 0F 38 F6 /r ADOX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

Instruction Operand Encoding

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

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is 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, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

```
IF OperandSize is 64-bit
    THEN OF:DEST[63:0] := DEST[63:0] + SRC[63:0] + OF;
    ELSE OF:DEST[31:0] := DEST[31:0] + SRC[31:0] + OF;
FI;
```

Flags Affected

OF is updated based on result. CF, SF, ZF, AF, and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

```
unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#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.
#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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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	A	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	C	V/V	VAES AVX512VL	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	C	V/V	VAES AVX512VL	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DE /r VAESDEC zmm1, zmm2, zmm3/m512	C	V/V	VAES AVX512F	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

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

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDEC

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

VAESDEC (128b and 256b VEX Encoded Versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

VAESDEC (EVEX Encoded Version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)
VAESDEC __m256i _mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC __m512i _mm512_aesdec_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-50, “Type E4NF Class Exception Conditions.”

AESDEC128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DD !{(11);rrr:bbb AESDEC128KL xmm, m384	A	V/V	AESKLE	Decrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

The AESDEC128KL¹ instruction performs 10 rounds of AES to decrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDEC128KL

```

Handle := UnalignedLoad of 384 bit (SRC);    // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [2] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);
IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], lWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
            ELSE
                DEST := AES128Decrypt (DEST, UnwrappedKey);
                RFLAGS.ZF := 0;
        FI;
    FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESDEC128KL    unsigned char _mm_aesdec128kl_u8(__m128i* odata, __m128i idata, const void* h);
```

1. Further details on Key Locker and usage of this instruction can be found here:
<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.07H:ECX.KL [bit 23] = 0.</p> <p>If CR4.KL = 0.</p> <p>If CPUID.19H:EBX.AESKLE [bit 0] = 0.</p> <p>If CR0.EM = 1.</p> <p>If CR4.OSFXSR = 0.</p>
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If the memory address is in a non-canonical form.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If a memory address referencing the SS segment is in a non-canonical form.</p>

AESDEC256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DF !{11};rrr:bbb AESDEC256KL xmm, m512	A	V/V	AESKLE	Decrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

The AESDEC256KL¹ instruction performs 14 rounds of AES to decrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDEC256KL

```

Handle := UnalignedLoad of 512 bit (SRC);    // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [2] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], lWKey);
    IF (Authentic == 0)
        THEN RFLAGS.ZF := 1;
    ELSE
        DEST := AES256Decrypt (DEST, UnwrappedKey);
        RFLAGS.ZF := 0;
FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESDEC256KL    unsigned char _mm_aesdec256kl_u8(__m128i* odata, __m128i idata, const void* h);
```

1. Further details on Key Locker and usage of this instruction can be found here:
<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.07H:ECX.KL [bit 23] = 0.</p> <p>If CR4.KL = 0.</p> <p>If CPUID.19H:EBX.AESKLE [bit 0] = 0.</p> <p>If CR0.EM = 1.</p> <p>If CR4.OSFXSR = 0.</p>
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If the memory address is in a non-canonical form.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If a memory address referencing the SS segment is in a non-canonical form.</p>

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	A	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES AVX512VL	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES AVX512VL	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES AVX512F	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDECLAST

```

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

```

VAESDECLAST (128b and 256b VEX Encoded Versions)

```

(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

```

VAESDECLAST (EVEX Encoded Version)

```

(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

```

Intel C/C++ Compiler Intrinsic Equivalent

```

(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)
VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-50, “Type E4NF Class Exception Conditions.”

AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !{11}:001:bbb AESDECWIDE128KL m384, <XMM0-7>	A	V/V	AESKLEWIDE_KL	Decrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESDECWIDE128KL¹ instruction performs ten rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDECWIDE128KL

```

Handle := UnalignedLoad of 384 bit (SRC);    // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [2] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], lWKey);
    IF Authentic == 0 {
        THEN RFLAGS.ZF := 1;
        ELSE
            XMM0 := AES128Decrypt (XMM0, UnwrappedKey);
            XMM1 := AES128Decrypt (XMM1, UnwrappedKey);
            XMM2 := AES128Decrypt (XMM2, UnwrappedKey);
            XMM3 := AES128Decrypt (XMM3, UnwrappedKey);
            XMM4 := AES128Decrypt (XMM4, UnwrappedKey);
            XMM5 := AES128Decrypt (XMM5, UnwrappedKey);
            XMM6 := AES128Decrypt (XMM6, UnwrappedKey);
            XMM7 := AES128Decrypt (XMM7, UnwrappedKey);
            RFLAGS.ZF := 0;
        FI;
    FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESDECWIDE128KL unsigned char _mm_aesdecwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);
```

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.07H:ECX.KL [bit 23] = 0.</p> <p>If CR4.KL = 0.</p> <p>If CPUID.19H:EBX.AESKLE [bit 0] = 0.</p> <p>If CR0.EM = 1.</p> <p>If CR4.OSFXSR = 0.</p> <p>If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.</p>
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If the memory address is in a non-canonical form.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If a memory address referencing the SS segment is in a non-canonical form.</p>

AESDECWIDE256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !{11}:011:bbb AESDECWIDE256KL m512, <XMM0-7>	A	V/V	AESKLEWIDE_KL	Decrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESDECWIDE256KL¹ instruction performs 14 rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESDECWIDE256KL

Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.

Illegal Handle = (HandleReservedBitSet (Handle) ||

 (Handle[0] AND (CPL > 0)) ||

 Handle [2] ||

 HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);

IF (Illegal Handle) {

 THEN RFLAGS.ZF := 1;

ELSE

 (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], lWKey);

 IF (Authentic == 0)

 THEN RFLAGS.ZF := 1;

 ELSE

 XMM0 := AES256Decrypt (XMM0, UnwrappedKey);

 XMM1 := AES256Decrypt (XMM1, UnwrappedKey);

 XMM2 := AES256Decrypt (XMM2, UnwrappedKey);

 XMM3 := AES256Decrypt (XMM3, UnwrappedKey);

 XMM4 := AES256Decrypt (XMM4, UnwrappedKey);

 XMM5 := AES256Decrypt (XMM5, UnwrappedKey);

 XMM6 := AES256Decrypt (XMM6, UnwrappedKey);

 XMM7 := AES256Decrypt (XMM7, UnwrappedKey);

 RFLAGS.ZF := 0;

 FI;

FI;

RFLAGS.OF, SF, AF, PF, CF := 0;

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESDECWIDE256KL unsigned char _mm_aesdecwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);
```

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.07H:ECX.KL [bit 23] = 0.</p> <p>If CR4.KL = 0.</p> <p>If CPUID.19H:EBX.AESKLE [bit 0] = 0.</p> <p>If CR0.EM = 1.</p> <p>If CR4.OSFXSR = 0.</p> <p>If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.</p>
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If the memory address is in a non-canonical form.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If a memory address referencing the SS segment is in a non-canonical form.</p>

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	A	V/V	AES	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	C	V/V	VAES AVX512VL	Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	C	V/V	VAES AVX512VL	Perform one round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from the ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DC /r VAESENC zmm1, zmm2, zmm3/m512	C	V/V	VAES AVX512F	Perform one round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from the zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs a single round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores 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.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENC

```

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

```

VAESENCE (128b and 256b VEX Encoded Versions)

(KL,VL) = (1,128), (2,256)

FOR I := 0 to KL-1:

STATE := SRC1.xmm[i]

RoundKey := SRC2.xmm[i]

STATE := ShiftRows(STATE)

STATE := SubBytes(STATE)

STATE := MixColumns(STATE)

DEST.xmm[i] := STATE XOR RoundKey

DEST[MAXVL-1:VL] := 0

VAESENCE (EVEX Encoded Version)

(KL,VL) = (1,128), (2,256), (4,512)

FOR i := 0 to KL-1:

STATE := SRC1.xmm[i] // xmm[i] is the i'th xmm word in the SIMD register

RoundKey := SRC2.xmm[i]

STATE := ShiftRows(STATE)

STATE := SubBytes(STATE)

STATE := MixColumns(STATE)

DEST.xmm[i] := STATE XOR RoundKey

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCE __m128i _mm_aesenc (__m128i, __m128i)

VAESENCE __m256i _mm256_aesenc_epi128(__m256i, __m256i);

VAESENCE __m512i _mm512_aesenc_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."

AESENC128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DC !{11};rrr:bbb AESENC128KL xmm, m384	A	V/V	AESKLE	Encrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

The AESENC128KL¹ instruction performs ten rounds of AES to encrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENC128KL

Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.

```

Illegal Handle = (
    HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [1] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
);
IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], lWKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
        ELSE
            DEST := AES128Encrypt (DEST, UnwrappedKey);
            RFLAGS.ZF := 0;
        FI;
    FI;
    RFLAGS.OF, SF, AF, PF, CF := 0;

```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESENC128KL unsigned char _mm_aesenc128kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.07H:ECX.KL [bit 23] = 0.</p> <p>If CR4.KL = 0.</p> <p>If CPUID.19H:EBX.AESKLE [bit 0] = 0.</p> <p>If CR0.EM = 1.</p> <p>If CR4.OSFXSR = 0.</p>
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If the memory address is in a non-canonical form.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If a memory address referencing the SS segment is in a non-canonical form.</p>

AESENC256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DE <i>l</i> (11);rrr:bbb AESENC256KL <i>xmm</i> , <i>m512</i>	A	V/V	AESKLE	Encrypt <i>xmm</i> using 256-bit AES key indicated by handle at <i>m512</i> and store result in <i>xmm</i> .

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (<i>r</i> , <i>w</i>)	ModRM:r/m (<i>r</i>)	N/A	N/A

Description

The AESENC256KL¹ instruction performs 14 rounds of AES to encrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENC256KL

```

Handle := UnalignedLoad of 512 bit (SRC);    // Load is not guaranteed to be atomic.
Illegal Handle = (
    HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [1] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256
);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], lWKey);
    IF (Authentic == 0)
        THEN RFLAGS.ZF := 1;
    ELSE
        DEST := AES256Encrypt (DEST, UnwrappedKey);
        RFLAGS.ZF := 0;
FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESENC256KL unsigned char _mm_aesenc256kl_u8(__m128i* odata, __m128i idata, const void* h);
```

1. Further details on Key Locker and usage of this instruction can be found here:
<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Exceptions (All Operating Modes)

#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.07H:ECX.KL [bit 23] = 0.</p> <p>If CR4.KL = 0.</p> <p>If CPUID.19H:EBX.AESKLE [bit 0] = 0.</p> <p>If CR0.EM = 1.</p> <p>If CR4.OSFXSR = 0.</p>
#NM	If CR0.TS = 1.
#PF	If a page fault occurs.
#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If the memory address is in a non-canonical form.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p> <p>If a memory address referencing the SS segment is in a non-canonical form.</p>

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	A	V/V	AES	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.
VEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	B	V/V	AES AVX	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
VEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	B	V/V	VAES	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	C	V/V	VAES AVX512VL	Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.
EVEX.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	C	V/V	VAES AVX512VL	Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.
EVEX.512.66.0F38.WIG DD /r VAESENCLAST zmm1, zmm2, zmm3/m512	C	V/V	VAES AVX512F	Perform the last round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction performs the last round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENCLAST

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

VAESENCLAST (128b and 256b VEX Encoded Versions)

(KL, VL) = (1,128), (2,256)

FOR I=0 to KL-1:

```
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
```

DEST[MAXVL-1:VL] := 0

VAESENCLAST (EVEX Encoded Version)

(KL,VL) = (1,128), (2,256), (4,512)

FOR i = 0 to KL-1:

```
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := ShiftRows( STATE )
    STATE := SubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
```

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)AESENCLAST __m128i __mm_aesencast (__m128i, __m128i)
VAESENCLAST __m256i __mm256_aesencast_epi128(__m256i, __m256i);
VAESENCLAST __m512i __mm512_aesencast_epi128(__m512i, __m512i);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-50, “Type E4NF Class Exception Conditions.”

AESENCWIDE128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !{11}:000:bbb AESENCWIDE128KL m384, <XMM0-7>	A	V/V	AESKLE WIDE_KL	Encrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESENCWIDE128KL¹ instruction performs ten rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENCWIDE128KL

Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.

```

Illegal Handle = (
    HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [1] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], lWKey);
    IF Authentic == 0
        THEN RFLAGS.ZF := 1;
    ELSE
        XMM0 := AES128Encrypt (XMM0, UnwrappedKey);
        XMM1 := AES128Encrypt (XMM1, UnwrappedKey);
        XMM2 := AES128Encrypt (XMM2, UnwrappedKey);
        XMM3 := AES128Encrypt (XMM3, UnwrappedKey);
        XMM4 := AES128Encrypt (XMM4, UnwrappedKey);
        XMM5 := AES128Encrypt (XMM5, UnwrappedKey);
        XMM6 := AES128Encrypt (XMM6, UnwrappedKey);
        XMM7 := AES128Encrypt (XMM7, UnwrappedKey);
        RFLAGS.ZF := 0;

    FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESENCWIDE128KLunsigned char _mm_aesencwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);
```

Exceptions (All Operating Modes)

#UD	<div>If the LOCK prefix is used. If CPUID.07H:ECX.KL [bit 23] = 0. If CR4.KL = 0. If CPUID.AESKLE = 0. If CR0.EM = 1. If CR4.OSFXSR = 0. If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.</div>
#NM	<div>If CR0.TS = 1.</div>
#PF	<div>If a page fault occurs.</div>
#GP(0)	<div>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. If the memory address is in a non-canonical form.</div>
#SS(0)	<div>If a memory operand effective address is outside the SS segment limit. If a memory address referencing the SS segment is in a non-canonical form.</div>

AESENCWIDE256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !{11}:010:bbb AESENCWIDE256KL m512, <XMM0-7>	A	V/V	AESKLE WIDE_KL	Encrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2—9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Description

The AESENCWIDE256KL¹ instruction performs 14 rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENCWIDE256KL

Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.

```

Illegal Handle = (
    HandleReservedBitSet (Handle) ||
    (Handle[0] AND (CPL > 0)) ||
    Handle [1] ||
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256
);
IF (Illegal Handle)
    THEN RFLAGS.ZF := 1;
ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], lWKey);
    IF (Authentic == 0)
        THEN RFLAGS.ZF := 1;
    ELSE
        XMM0 := AES256Encrypt (XMM0, UnwrappedKey);
        XMM1 := AES256Encrypt (XMM1, UnwrappedKey);
        XMM2 := AES256Encrypt (XMM2, UnwrappedKey);
        XMM3 := AES256Encrypt (XMM3, UnwrappedKey);
        XMM4 := AES256Encrypt (XMM4, UnwrappedKey);
        XMM5 := AES256Encrypt (XMM5, UnwrappedKey);
        XMM6 := AES256Encrypt (XMM6, UnwrappedKey);
        XMM7 := AES256Encrypt (XMM7, UnwrappedKey);
        RFLAGS.ZF := 0;
    FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

```
AESENCWIDE256KLunsigned char _mm_aesencwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);
```

Exceptions (All Operating Modes)

#UD	<div>If the LOCK prefix is used. If CPUID.07H:ECX.KL [bit 23] = 0. If CR4.KL = 0. If CPUID.19H:EBX.AESKLE [bit 0] = 0. If CR0.EM = 1. If CR4.OSFXSR = 0. If CPUID.19H:EBX.WIDE_KL [bit 2] = 0.</div>
#NM	<div>If CR0.TS = 1.</div>
#PF	<div>If a page fault occurs.</div>
#GP(0)	<div>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. If the memory address is in a non-canonical form.</div>
#SS(0)	<div>If a memory operand effective address is outside the SS segment limit. If a memory address referencing the SS segment is in a non-canonical form.</div>

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.DB 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	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

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 (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-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] := InvMixColumns(SRC);

DEST[MAXVL-1:128] (Unmodified)

VAESIMC

DEST[127:0] := InvMixColumns(SRC);

DEST[MAXVL-1:128] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC __m128i __mm_aesimc (__m128i)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” 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	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	N/A

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 (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-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[MAXVL-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[MAXVL-1:128] := 0;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESKEYGENASSIST __m128i _mm_aeskeygenassist (__m128i, const int)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD If VEX.vvvv ≠ 1111B.

AND—Logical AND

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
24 ib	AND AL, imm8	I	Valid	Valid	AL AND imm8.
25 iw	AND AX, imm16	I	Valid	Valid	AX AND imm16.
25 id	AND EAX, imm32	I	Valid	Valid	EAX AND imm32.
REX.W + 25 id	AND RAX, imm32	I	Valid	N.E.	RAX AND imm32 sign-extended to 64-bits.
80 /4 ib	AND r/m8, imm8	MI	Valid	Valid	r/m8 AND imm8.
REX + 80 /4 ib	AND r/m8 [*] , imm8	MI	Valid	N.E.	r/m8 AND imm8.
81 /4 iw	AND r/m16, imm16	MI	Valid	Valid	r/m16 AND imm16.
81 /4 id	AND r/m32, imm32	MI	Valid	Valid	r/m32 AND imm32.
REX.W + 81 /4 id	AND r/m64, imm32	MI	Valid	N.E.	r/m64 AND imm32 sign extended to 64-bits.
83 /4 ib	AND r/m16, imm8	MI	Valid	Valid	r/m16 AND imm8 (sign-extended).
83 /4 ib	AND r/m32, imm8	MI	Valid	Valid	r/m32 AND imm8 (sign-extended).
REX.W + 83 /4 ib	AND r/m64, imm8	MI	Valid	N.E.	r/m64 AND imm8 (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)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A

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

DEST := 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.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.LZ.0F38.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)	N/A

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 Table 2-29, “Type 13 Class Exception Conditions.”

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VANDPD (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

 i := j * 64

 IF k1[j] OR *no writemask*

 THEN

 IF (EVEX.b == 1) AND (SRC2 *is memory*)

 THEN

 DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[63:0]

 ELSE

 DEST[i+63:i] := SRC1[i+63:i] BITWISE AND SRC2[i+63:i]

 FI;

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[i+63:i] remains unchanged*

 ELSE

 ; zeroing-masking

 DEST[i+63:i] = 0

 FI;

 FI;

ENDFOR

DEST[MAXVL-1:VL] := 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]

DEST[MAXVL-1:256] := 0

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[MAXVL-1:128] := 0

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[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDPD __m512d __mm512_and_pd (__m512d a, __m512d b);

VANDPD __m512d __mm512_mask_and_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);

VANDPD __m512d __mm512_maskz_and_pd (__mmask8 k, __m512d a, __m512d b);

VANDPD __m256d __mm256_mask_and_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);

VANDPD __m256d __mm256_maskz_and_pd (__mmask8 k, __m256d a, __m256d b);

VANDPD __m128d __mm_mask_and_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);

VANDPD __m128d __mm_maskz_and_pd (__mmask8 k, __m128d a, __m128d b);

VANDPD __m256d __mm256_and_pd (__m256d a, __m256d b);

ANDPD __m128d __mm_and_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.OF 54 /r VANDPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.OF 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.OF.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.OF.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.OF.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDPS (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

 i := j * 32

 IF k1[j] OR *no writemask*

 IF (EVEX.b == 1) AND (SRC2 *is memory*)

 THEN

 DEST[i+63:i] := SRC1[i+31:i] BITWISE AND SRC2[31:0]

 ELSE

 DEST[i+31:i] := SRC1[i+31:i] BITWISE AND SRC2[i+31:i]

 FI;

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[i+31:i] remains unchanged*

 ELSE ; zeroing-masking

 DEST[i+31:i] := 0

 FI;

 FI;

ENDFOR

DEST[MAXVL-1:VL] := 0;

VANDPS (VEX.256 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[159:128] := SRC1[159:128] BITWISE AND SRC2[159:128]

DEST[191:160] := SRC1[191:160] BITWISE AND SRC2[191:160]

DEST[223:192] := SRC1[223:192] BITWISE AND SRC2[223:192]

DEST[255:224] := SRC1[255:224] BITWISE AND SRC2[255:224].

DEST[MAXVL-1:256] := 0;

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[MAXVL-1:128] := 0;

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[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

```

VANDPS __m512 __mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 __mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 __mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 __mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 __mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 __mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 __mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 __mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 __mm_and_ps (__m128 a, __m128 b);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical AND NOT of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VANDNPD (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

 i := j * 64

 IF k1[j] OR *no writemask*

 IF (EVEX.b == 1) AND (SRC2 *is memory*)

 THEN

 DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]

 ELSE

 DEST[i+63:i] := (NOT(SRC1[i+63:i])) BITWISE AND SRC2[i+63:i]

 FI;

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[i+63:i] remains unchanged*

 ELSE ; zeroing-masking

 DEST[i+63:i] = 0

 FI;

 FI;

ENDFOR

DEST[MAXVL-1:VL] := 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]

DEST[MAXVL-1:256] := 0

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[MAXVL-1:128] := 0

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[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD __m512d __mm512_andnot_pd (__m512d a, __m512d b);

VANDNPD __m512d __mm512_mask_andnot_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);

VANDNPD __m512d __mm512_maskz_andnot_pd (__mmask8 k, __m512d a, __m512d b);

VANDNPD __m256d __mm256_mask_andnot_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);

VANDNPD __m256d __mm256_maskz_andnot_pd (__mmask8 k, __m256d a, __m256d b);

VANDNPD __m128d __mm_mask_andnot_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);

VANDNPD __m128d __mm_maskz_andnot_pd (__mmask8 k, __m128d a, __m128d b);

VANDNPD __m256d __mm256_andnot_pd (__m256d a, __m256d b);

ANDNPD __m128d __mm_andnot_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical AND NOT of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDNPS (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

 i := j * 32

 IF k1[j] OR *no writemask*

 IF (EVEX.b == 1) AND (SRC2 *is memory*)

 THEN

 DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]

 ELSE

 DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]

 FI;

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[i+31:i] remains unchanged*

 ELSE ; zeroing-masking

 DEST[i+31:i] = 0

 FI;

 FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VANDNPS (VEX.256 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[159:128] := (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]

DEST[191:160] := (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]

DEST[223:192] := (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]

DEST[255:224] := (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].

DEST[MAXVL-1:256] := 0

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[MAXVL-1:128] := 0

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[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

```

VANDNPS __m512 __mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 __mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 __mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 __mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 __mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 __mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 __mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 __mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 __mm_andnot_ps (__m128 a, __m128 b);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
63 /r	ARPL r/m16, r16	MR	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
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

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 MOVSLD;

ELSE

IF DEST[RPL] < SRC[RPL]

THEN

ZF := 1;

DEST[RPL] := SRC[RPL];

ELSE

ZF := 0;

FI;

FI;

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.

BEXTR—Bit Field Extract

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.OF38.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.LZ.OF38.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.

Instruction Operand Encoding

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

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 Table 2-29, "Type 13 Class Exception Conditions," additionally:

#UD If VEX.W = 1.

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 xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Select packed double precision floating-point values from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.
VEX.128.66.0F3A.WIG 0D /r ib VBLENDPD xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Select packed double precision floating-point Values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1.
VEX.256.66.0F3A.WIG 0D /r ib VBLENDPD ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Select packed double precision floating-point Values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
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 (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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 Table 2-21, "Type 4 Class Exception Conditions."

BLENDPS—Blend Packed Single Precision Floating-Point Values

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
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 (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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 Table 2-21, "Type 4 Class Exception Conditions."

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 xmm1, xmm2/m128, <XMM0>	RM0	V/V	SSE4_1	Select packed double precision floating-point values from xmm1 and xmm2 from mask specified in XMM0 and store the values in xmm1.
VEX.128.66.0F3A.W0 4B /r /is4 VBLENDVPD xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Conditionally copy double precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4.
VEX.256.66.0F3A.W0 4B /r /is4 VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX	Conditionally copy double precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	N/A
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 (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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 Table 2-21, “Type 4 Class Exception Conditions,” 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 xmm1, xmm2/m128, <XMM0>	RMO	V/V	SSE4_1	Select packed single precision floating-point values from xmm1 and xmm2/m128 from mask specified in XMM0 and store the values into xmm1.
VEX.128.66.0F3A.W0 4A /r /is4 VBLENDVPS xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Conditionally copy single precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the specified mask operand, xmm4.
VEX.256.66.0F3A.W0 4A /r /is4 VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX	Conditionally copy single precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the specified mask register, ymm4.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMO	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	N/A
RVMR	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 (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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 Table 2-21, “Type 4 Class Exception Conditions,” 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.LZ.OF38.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.LZ.OF38.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)	N/A	N/A

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 _bsl_u32(unsigned __int32 src);
BLSI unsigned __int64 _bsl_u64(unsigned __int64 src);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, “Type 13 Class Exception Conditions.”

BLSMSK—Get Mask Up to Lowest Set Bit

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.OF38.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.LZ.OF38.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)	N/A	N/A

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 if 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 Table 2-29, “Type 13 Class Exception Conditions.”

BLSR—Reset Lowest Set Bit

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.LZ.OF38.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.LZ.OF38.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)	N/A	N/A

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 Table 2-29, “Type 13 Class Exception Conditions.”

BNDCL—Check Lower Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 1A /r BNDCL bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.
F3 OF 1A /r BNDCL bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.

Instruction Operand Encoding

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

Description

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

Operation

BNDCL BND, reg

```
IF reg < BND.LB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BNDCL BND, mem

```
TEMP := LEA(mem);
IF TEMP < BND.LB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
BNDCL void _bnd_chk_ptr_lbounds(const void *q)
```

Flags Affected

None

Protected Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
-----	--

Same exceptions as in protected mode.

BND_{CU}/BND_{CN}—Check Upper Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 1A /r BND _{CU} bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 OF 1A /r BND _{CU} bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 OF 1B /r BND _{CN} bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).
F2 OF 1B /r BND _{CN} bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).

Instruction Operand Encoding

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

Description

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BND_{CU} perform 1's complement operation on the upper bound of bnd first before proceeding with address comparison. BND_{CN} perform address comparison directly using the upper bound in bnd that is already reverted out of 1's complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

Operation

BND_{CU} BND, reg

```
IF reg > NOT(BND.UB) Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BND_{CU} BND, mem

```
TEMP := LEA(mem);
IF TEMP > NOT(BND.UB) Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

BND_{CN} BND, reg

```
IF reg > BND.UB Then
    BNDSTATUS := 01H;
    #BR;
FI;
```

```

BND CN BND, mem
TEMP := LEA(mem);
IF TEMP > BND.UB Then
    BNDSTATUS := 01H;
    #BR;
FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
BND CU .void _bnd_chk_ptr_ubounds(const void *q)
```

Flags Affected

None

Protected Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
-----	--

Same exceptions as in protected mode.

BNDLDX—Load Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	N/A

Description

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

base := mib.SIB.base ? mib.SIB.base + Disp: 0;

ptr_value := mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit Mode

A_BDE[31:0] := (Zero_extend32(base[31:12] << 2) + (BNDCFG[31:12] << 12));

A_BT[31:0] := LoadFrom(A_BDE);

IF A_BT[0] equal 0 Then

 BNDSTATUS := A_BDE | 02H;

 #BR;

FI;

A_BTE[31:0] := (Zero_extend32(base[11:2] << 4) + (A_BT[31:2] << 2));

Temp_lb[31:0] := LoadFrom(A_BTE);

Temp_ub[31:0] := LoadFrom(A_BTE + 4);

Temp_ptr[31:0] := LoadFrom(A_BTE + 8);

IF Temp_ptr equal ptr_value Then

 BND.LB := Temp_lb;

 BND.UB := Temp_ub;

```

ELSE
    BND.LB := 0;
    BND.UB := 0;
FI;

```

In 64-bit Mode

```

A_BDE[63:0] := (Zero_extend64(base[47+MAWA:20] << 3) + (BNDCFG[63:12] << 12));1
A_BT[63:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] := (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3);
Temp_lb[63:0] := LoadFrom(A_BTE);
Temp_ub[63:0] := LoadFrom(A_BTE + 8);
Temp_ptr[63:0] := LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB := Temp_lb;
    BND.UB := Temp_ub;
ELSE
    BND.LB := 0;
    BND.UB := 0;
FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit. If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Appendix E.3.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative. If the LOCK prefix is used. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code)	If a page fault occurs.

BNDMK—Make Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 OF 1B /r BNDMK bnd, m32	RM	NE/V	MPX	Make lower and upper bounds from m32 and store them in bnd.
F3 OF 1B /r BNDMK bnd, m64	RM	V/NE	MPX	Make lower and upper bounds from m64 and store them in bnd.

Instruction Operand Encoding

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

Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

Operation

```

BND.LB := SRCMEM.base;
IF 64-bit mode Then
    BND.UB := NOT(LEA.64_bits(SRCMEM));
ELSE
    BND.UB := Zero_Extend.64_bits(NOT(LEA.32_bits(SRCMEM)));
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
BNDMKvoid * _bnd_set_ptr_bounds(const void * q, size_t size);
```

Flags Affected

None.

Protected Mode Exceptions

#UD

- If the LOCK prefix is used.
- If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
- If 67H prefix is not used and CS.D=0.
- If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#UD

- If the LOCK prefix is used.
- If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
- If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
 If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
 If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
 If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
 If RIP-relative addressing is used.
#SS(0) If the memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.

Same exceptions as in protected mode.

BNDMOV—Move Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 OF 1A /r BNDMOV bnd1, bnd2/m64	RM	NE/V	MPX	Move lower and upper bound from bnd2/m64 to bound register bnd1.
66 OF 1A /r BNDMOV bnd1, bnd2/m128	RM	V/NE	MPX	Move lower and upper bound from bnd2/m128 to bound register bnd1.
66 OF 1B /r BNDMOV bnd1/m64, bnd2	MR	NE/V	MPX	Move lower and upper bound from bnd2 to bnd1/m64.
66 OF 1B /r BNDMOV bnd1/m128, bnd2	MR	V/NE	MPX	Move lower and upper bound from bnd2 to bound register bnd1/m128.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 3-5.

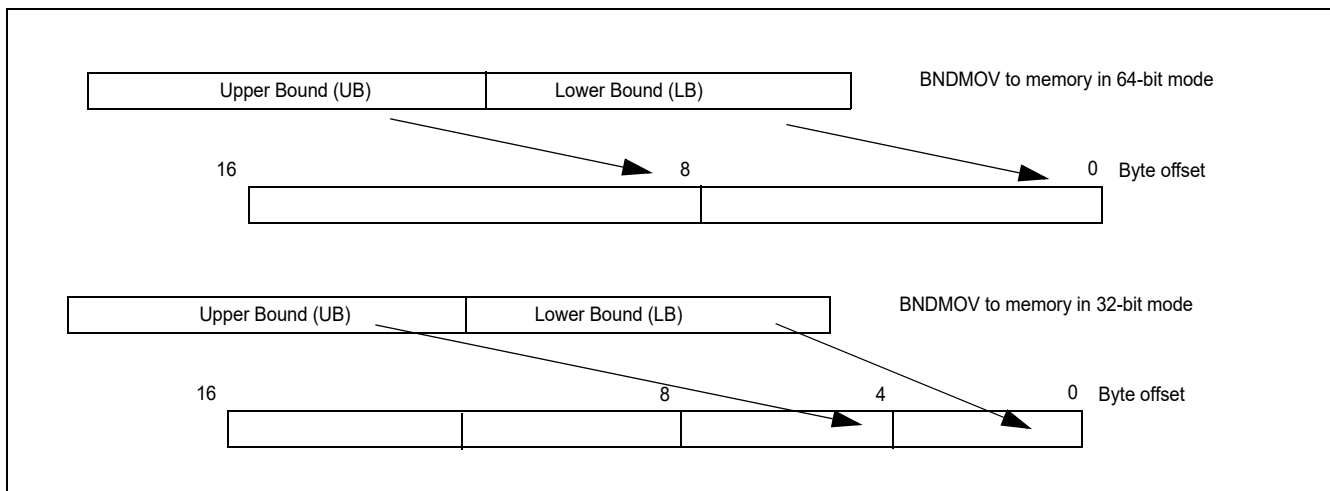


Figure 3-5. Memory Layout of BNDMOV to/from Memory

This instruction does not change flags.

Operation

BNDMOV register to register

DEST.LB := SRC.LB;

DEST.UB := SRC.UB;

BNDMOV from memory

```

IF 64-bit mode THEN
    DEST.LB := LOAD_QWORD(SRC);
    DEST.UB := LOAD_QWORD(SRC+8);
ELSE
    DEST.LB := LOAD_DWORD_ZERO_EXT(SRC);
    DEST.UB := LOAD_DWORD_ZERO_EXT(SRC+4);
FI;

```

BNDMOV to memory

```

IF 64-bit mode THEN
    DEST[63:0] := SRC.LB;
    DEST[127:64] := SRC.UB;
ELSE
    DEST[31:0] := SRC.LB;
    DEST[63:32] := SRC.UB;
FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
BNDMOV void * _bnd_copy_ptr_bounds(const void *q, const void *r)
```

Flags Affected

None.

Protected Mode Exceptions

#UD	<p>If the LOCK prefix is used but the destination is not a memory operand.</p> <p>If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.</p> <p>If 67H prefix is not used and CS.D=0.</p> <p>If 67H prefix is used and CS.D=1.</p>
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#GP(0)	<p>If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the destination operand points to a non-writable segment</p> <p>If the DS, ES, FS, or GS segment register contains a NULL segment selector.</p>
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	<p>If the LOCK prefix is used but the destination is not a memory operand.</p> <p>If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.</p> <p>If 16-bit addressing is used.</p>
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If the memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	N/A

Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

base := mib.SIB.base ? mib.SIB.base + Disp: 0;

ptr_value := mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit Mode

A_BDE[31:0] := (Zero_extend32(base[31:12] << 2) + (BNDCFG[31:12] << 12));

A_BT[31:0] := LoadFrom(A_BDE);

IF A_BT[0] equal 0 Then

 BNDSTATUS := A_BDE | 02H;

 #BR;

FI;

A_DEST[31:0] := (Zero_extend32(base[31:12] << 4) + (A_BT[31:2] << 2)); // address of Bound table entry

A_DEST[8][31:0] := ptr_value;

A_DEST[0][31:0] := BND.LB;

A_DEST[4][31:0] := BND.UB;

In 64-bit Mode

```

A_BDE[63:0] := (Zero_extend64(base[47+MAWA:20] << 3) + (BNDCFG[63:12] << 12));1
A_BT[63:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] := (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3)); // address of Bound table entry
A_DEST[16][63:0] := ptr_value;
A_DEST[0][63:0] := BND.LB;
A_DEST[8][63:0] := BND.UB;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);
```

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit. If DS register contains a NULL segment selector. If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Appendix E.3.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative. If the LOCK prefix is used. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form. If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

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 r16 (array index) is within bounds specified by m16&16.
62 /r	BOUND r32, m32&32	RM	Invalid	Valid	Check if r32 (array index) is within bounds specified by m32&32.

Instruction Operand Encoding

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

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) THEN
      (* Below lower bound or above upper bound *)
      IF <equation for PL enabled> THEN BNDSTATUS := 0
      #BR;
    FI;
  FI;
```

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)	N/A	N/A

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 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)	N/A	N/A

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 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+ <i>rd</i>	BSWAP <i>r32</i>	0	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + OF C8+ <i>rd</i>	BSWAP <i>r64</i>	0	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 + <i>rd</i> (<i>r</i> , <i>w</i>)	N/A	N/A	N/A

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];
```

FI;

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)	N/A	N/A
MI	ModRM:r/m (r)	imm8	N/A	N/A

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 3-11.

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:

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

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

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

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 := Bit(BitBase, 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 r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and complement.
OF BB /r	BTC r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BB /r	BTC r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and complement.
OF BA /7 ib	BTC r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and complement.
OF BA /7 ib	BTC r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BA /7 ib	BTC r/m64, imm8	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)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

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 3-11.

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)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

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 3-11.

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 r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and set.
OF AB /r	BTS r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF AB /r	BTS r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and set.
OF BA /5 ib	BTS r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and set.
OF BA /5 ib	BTS r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF BA /5 ib	BTS r/m64, imm8	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)	N/A	N/A
MI	ModRM:r/m (r, w)	imm8	N/A	N/A

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 3-11.

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.LZ.OF38.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.LZ.OF38.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.

Instruction Operand Encoding

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

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 and SF flags are updated based on the result. CF flag is set as specified in the Operation section. 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 Table 2-29, "Type 13 Class Exception Conditions."

CALL—Call Procedure

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
E8 cw	CALL rel16	D	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 cd	CALL rel32	D	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 r/m16	M	N.E.	Valid	Call near, absolute indirect, address given in r/m16.
FF /2	CALL r/m32	M	N.E.	Valid	Call near, absolute indirect, address given in r/m32.
FF /2	CALL r/m64	M	Valid	N.E.	Call near, absolute indirect, address given in r/m64.
9A cd	CALL ptr16:16	D	Invalid	Valid	Call far, absolute, address given in operand.
9A cp	CALL ptr16:32	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL m16:16	M	Valid	Valid	Call far, absolute indirect address given in m16:16. 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 m16:32	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 m16:64	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	N/A	N/A	N/A
M	ModRM:r/m (r)	N/A	N/A	N/A

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 8, “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 (*rel16* or *rel32*) 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 procedure's 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 8, "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 8 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 22, "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.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 17, “Control-Flow Enforcement Technology (CET)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for CET details.

Instruction ordering. Instructions following a far call may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far call have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Instructions sequentially following a near indirect CALL instruction (i.e., those not at the target) may be executed speculatively. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an LFENCE instruction opcode can be placed after the near indirect CALL in order to block speculative execution.

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); FI;
          Push(RIP);
          IF ShadowStackEnabled(CPL) AND DEST != 0
            ShadowStackPush8B(RIP);
          FI;
          RIP := tempRIP;
        FI;
      IF OperandSize = 32
        THEN
          tempEIP := EIP + DEST; (* DEST is rel32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); FI;
          Push(EIP);
          IF ShadowStackEnabled(CPL) AND DEST != 0
            ShadowStackPush4B(EIP);
          FI;
          EIP := tempEIP;
        FI;
      IF OperandSize = 16
        THEN
          tempEIP := (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 2-byte return address
            THEN #SS(0); FI;
          Push(IP);
          IF ShadowStackEnabled(CPL) AND DEST != 0
            (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
            ShadowStackPush4B(IP);
          FI;
          EIP := tempEIP;
        FI;
      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); FI;
        Push(RIP);
        IF ShadowStackEnabled(CPL)
            ShadowStackPush8B(RIP);
        FI;
        RIP := tempRIP;
    FI;
    IF OperandSize = 32
        THEN
            tempEIP := DEST; (* DEST is r/m32 *)
            IF tempEIP is not within code segment limit THEN #GP(0); FI;
            IF stack not large enough for a 4-byte return address
                THEN #SS(0); FI;
            Push(EIP);
            IF ShadowStackEnabled(CPL)
                ShadowStackPush4B(EIP);
            FI;
            EIP := tempEIP;
        FI;
    IF OperandSize = 16
        THEN
            tempEIP := DEST AND 0000FFFFH; (* DEST is r/m16 *)
            IF tempEIP is not within code segment limit THEN #GP(0); FI;
            IF stack not large enough for a 2-byte return address
                THEN #SS(0); FI;
            Push(IP);
            IF ShadowStackEnabled(CPL)
                (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
                ShadowStackPush4B(IP);
            FI;
            EIP := tempEIP;
        FI;
    FI;rel/abs
    IF (Call near indirect, absolute indirect)
        IF EndbranchEnabledAndNotSuppressed(CPL)
            IF CPL = 3
                THEN
                    IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                        FI;
                    ELSE
                        IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                            THEN
                                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                            FI;
                        FI;
                    FI;
                FI;
            FI;
        FI;
    FI; 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); FI;
        IF DEST[31:16] is not zero THEN #GP(0); FI;
        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); FI;
        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 *)
    FI;
FI;

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); FI;
    IF segment selector index not within descriptor table limits
    THEN #GP(new code segment selector); FI;
    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); FI;
    ELSE
        IF segment type is not a conforming or nonconforming code segment or
        64-bit 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;
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(new code segment selector); FI;
    IF segment not present
    THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address

```

```

    THEN #SS(0); FI;
tempEIP := DEST(Offset);
IF target mode = Compatibility mode
    THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
    THEN
        tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF OperandSize = 32
        THEN
            tempPushLIP = CSBASE + EIP;
        ELSE
            IF OperandSize = 16
                THEN
                    tempPushLIP = CSBASE + IP;
                ELSE (* OperandSize = 64 *)
                    tempPushLIP = RIP;
            FI;
        FI;
    tempPushCS = CS;
FI;
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;
            FI;
        FI;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
            (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)

```



```

        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
FI;
(* align to 8 byte boundary if not already aligned *)
tempSSP = SSP;
Shadow_stack_store 4 bytes of 0 to (SSP - 4)
SSP = SSP & 0xFFFFFFFFFFFFFFF8H
ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(tempPushLIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL ≠ CPL)
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP := DEST(Offset);
    IF target mode = Compatibility mode
        THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
        IF IA32_EFER.LMA & CS.L
            tempPushLIP = RIP
        ELSE
            tempPushLIP = CSBASE + EIP;
        FI;
        tempPushCS = CS;
    FI;
    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;
    FI;
FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
        FI;
    (* align to 8 byte boundary if not already aligned *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)
    SSP = SSP & 0xFFFFFFFFFFFFFFF8H
    ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
    ShadowStackPush8B(tempPushLIP); (* Padded 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled(CPL)
    IF CPL = 3
    THEN
        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_U_CET.SUPPRESS = 0
    ELSE
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

CALL-GATE:
    IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); FI;
    IF call gate not present
    THEN #NP(call-gate selector); FI;
    IF call-gate code-segment selector is NULL
    THEN #GP(0); FI;

```

```

IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); FI;
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); FI;
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); FI;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); FI;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
FI;
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); FI;
        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); FI;
                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); FI;
                NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
    FI;
IF IA32_EFER.LMA = 0 and NewSS is NULL
    THEN #TS(NewSS); FI;
Read 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); FI;
IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); FI;
IF CallGateSize = 32
    THEN
        IF new stack does not have room for parameters plus 16 bytes
            THEN #SS(NewSS); FI;
    FI;

```

```

    IF CallGate(InstructionPointer) not within new code-segment limit
        THEN #GP(0); FI;
    SS := newSS; (* Segment descriptor information also loaded *)
    ESP := newESP;
    CS:EIP := 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
    IF CallGateSize = 16
        THEN
            IF new stack does not have room for parameters plus 8 bytes
                THEN #SS(NewSS); FI;
            IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                THEN #GP(0); FI;
            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); FI;
            IF (CallGate(InstructionPointer) is non-canonical)
                THEN #GP(0); FI;
            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 *)
        FI;
    FI;
IF ShadowStackEnabled(CPL) AND CPL = 3
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
        ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
            IA32_PL3_SSP := LA_adjust(SSP);
        FI;
    FI;
CPL := CodeSegment(DPL)
CS(RPL) := CPL
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := IA32_PLi_SSP; (* where i is the CPL *)
    IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then #GP *)
        THEN #GP(0); FI;
    (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region*)

```

```

IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
    #GP(0); FI;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;
expected_token_value = SSP          (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT    (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
    THEN #GP(0); FI;
IF oldSS.DPL != 3
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(oldSSP);
FI;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;
END;

SAME-PRIVILEGE:
IF CallGateSize = 32
    THEN
        IF stack does not have room for 8 bytes
            THEN #SS(0); FI;
        IF CallGate(InstructionPointer) not within code segment limit
            THEN #GP(0); FI;
        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); FI;
                IF CallGate(InstructionPointer) not within code segment limit
                    THEN #GP(0); FI;
                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); FI;
                IF RIP non-canonical
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:instruction pointer);
                (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
        FI;
    FI;
FI;
CS(RPL) := CPL
IF ShadowStackEnabled(CPL)
    (* Align to next 8 byte boundary *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)

```

```

    SSP = SSP & 0xFFFFFFFFFFFFF8H;
    (* push cs:lip:ssp on shadow stack *)
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

```

TASK-GATE:

```

    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    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); FI;
    Access TSS descriptor in GDT;
    IF descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

TASK-STATE-SEGMENT:

```

    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
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)	<p>If the target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the gate is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p> <p>If SSP in IA32_PLI_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.</p> <p>If “supervisor Shadow Stack” token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.</p> <p>If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLI_SSP (where i is the new CPL).</p>
#GP(selector)	<p>If a code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>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.</p> <p>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.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>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.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>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.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>
#SS(selector)	<p>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.</p> <p>If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.</p> <p>If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.</p>
#NP(selector)	<p>If a code segment, data segment, call gate, task gate, or TSS is not present.</p>

#TS(selector)	<p>If the new stack segment selector and ESP are beyond the end of the TSS.</p> <p>If the new stack segment selector is NULL.</p> <p>If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.</p> <p>If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.</p> <p>If the new stack segment is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#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	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
#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)	<p>If a memory address is non-canonical.</p> <p>If target offset in destination operand is non-canonical.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the 64-bit gate is NULL.</p> <p>If target mode is compatibility mode and SSP is not in low 4GB.</p> <p>If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.</p> <p>If "supervisor Shadow Stack" token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit mode or compatibility mode, but SSP address in "super-visor shadow" stack token is beyond 4GB.</p> <p>If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).</p>
--------	---

#GP(selector)	<p>If code segment or 64-bit call gate is outside descriptor table limits.</p> <p>If code segment or 64-bit call gate overlaps non-canonical space.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.</p> <p>If the upper type field of a 64-bit call gate is not 0x0.</p> <p>If the segment selector from a 64-bit call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.</p> <p>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.</p> <p>If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.</p>
#SS(0)	<p>If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.</p> <p>If a memory operand effective address is outside the SS segment limit.</p> <p>If the stack address is in a non-canonical form.</p>
#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	<p>(64-bit mode only) If a far call is direct to an absolute address in memory.</p> <p>If the LOCK prefix is used.</p>
#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	Z0	Valid	Valid	AX := sign-extend of AL.
98	CWDE	Z0	Valid	Valid	EAX := sign-extend of AX.
REX.W + 98	CDQE	Z0	Valid	N.E.	RAX := sign-extend of EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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); FI;

ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)

RAX := SignExtend(EAX);

FI;

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLAC—Clear AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CA CLAC	Z0	V/V	SMAP	Clear the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

Operation

EFLAGS.AC := 0;

Flags Affected

AC cleared. Other flags are unaffected.

Protected Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD
 If the LOCK prefix is used.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

#UD The CLAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD
 If the LOCK prefix is used.
 If the CPL > 0.
 If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

CLC—Clear Carry Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
F8	CLC	Z0	Valid	Valid	Clear CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

Description

Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

CF := 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	Z0	Valid	Valid	Clear DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 modes.

Operation

DF := 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.

CLDEMOT—Cache Line Demote

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 1C /0 CLDEMOT m8	A	V/V	CLDEMOT	Hint to hardware to move the cache line containing m8 to a more distant level of the cache without writing back to memory.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (w)	N/A	N/A	N/A

Description

Hints to hardware that the cache line that contains the linear address specified with the memory operand should be moved (“demoted”) from the cache(s) closest to the processor core to a level more distant from the processor core. This may accelerate subsequent accesses to the line by other cores in the same coherence domain, especially if the line was written by the core that demotes the line. Moving the line in such a manner is a performance optimization, i.e., it is a hint which does not modify architectural state. Hardware may choose which level in the cache hierarchy to retain the line (e.g., L3 in typical server designs). The source operand is a byte memory location.

The availability of the CLDEMOT instruction is indicated by the presence of the CPUID feature flag CLDEMOT (bit 25 of the ECX register in sub-leaf 07H, see “CPUID—CPU Identification”). On processors which do not support the CLDEMOT instruction (including legacy hardware) the instruction will be treated as a NOP.

A CLDEMOT instruction is ordered with respect to stores to the same cache line, but unordered with respect to other instructions including memory fences, CLDEMOT, CLWB or CLFLUSHOPT instructions to a different cache line. Since CLDEMOT will retire in order with respect to stores to the same cache line, software should ensure that after issuing CLDEMOT the line is not accessed again immediately by the same core to avoid cache data movement penalties.

The effective memory type of the page containing the affected line determines the effect; cacheable types are likely to generate a data movement operation, while uncacheable types may cause the instruction to be ignored.

Speculative fetching can occur at any time and is not tied to instruction execution. The CLDEMOT instruction is not ordered with respect to PREFETCHH 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 CLDEMOT instruction that references the cache line.

Unlike CLFLUSH, CLFLUSHOPT, and CLWB instructions, CLDEMOT is not guaranteed to write back modified data to memory.

The CLDEMOT instruction may be ignored by hardware in certain cases and is not a guarantee.

The CLDEMOT instruction can be used at all privilege levels. In certain processor implementations the CLDEMOT instruction may set the A bit but not the D bit in the page tables.

If the line is not found in the cache, the instruction will be treated as a NOP.

In some implementations, the CLDEMOT instruction may always cause a transactional abort with Transactional Synchronization Extensions (TSX). However, programmers must not rely on CLDEMOT instruction to force a transactional abort.

1. The Mod field of the ModR/M byte cannot have value 11B.

Operation

Cache_Line_Demote(m8);

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

CLDEMOTE void _cldemote(const void*);

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

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.

CLFLUSH—Flush Cache Line

Opcode / Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
NP 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)	N/A	N/A	N/A

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (CPUID.01H:EDX[bit 19]). 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 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 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).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, and fence instructions.¹ They are not ordered with respect to executions of CLFLUSHOPT and CLWB. Software can use the SFENCE instruction to order an execution of CLFLUSH relative to one of those operations.

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.

In some implementations, the CLFLUSH instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

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)

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

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.

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.

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.

CLFLUSHOPT—Flush Cache Line Optimized

Opcode / Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
NFx 66 0F AE /7 CLFLUSHOPT m8	M	Valid	Valid	Flushes cache line containing m8.

Instruction Operand Encoding

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

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSHOPT is indicated by the presence of the CPUID feature flag CLFLUSHOPT (CPUID.(EAX=7,ECX=0):EBX[bit 23]). 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 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 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).

Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to older writes to the cache line being invalidated. They are not ordered with respect to other executions of CLFLUSHOPT, to executions of CLFLUSH and CLWB, or to younger writes to the cache line being invalidated. Software can use the SFENCE instruction to order an execution of CLFLUSHOPT relative to one of those operations.

The CLFLUSHOPT 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 CLFLUSHOPT instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSHOPT instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSHOPT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSHOPT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSHOPT instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

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

Operation

Flush_Cache_Line_Optimized(SRC);

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSHOPT void _mm_clflushopt(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.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0. If the LOCK prefix is used. If an instruction prefix F2H or F3H is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0. If the LOCK prefix is used. If an instruction prefix F2H or F3H 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.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0. If the LOCK prefix is used. If an instruction prefix F2H or F3H is used.

CLI—Clear Interrupt Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
FA	CLI	Z0	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

Description

In most cases, CLI clears the IF flag in the EFLAGS register and 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 effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

Table 3-7. Decision Table for CLI Results

Mode	IOPL	CLI Result
Real-address	X ¹	IF = 0
Protected, not PVI ²	≥ CPL	IF = 0
	< CPL	#GP fault
Protected, PVI ³	3	IF = 0
	0-2	VIF = 0
Virtual-8086, not VME ³	3	IF = 0
	0-2	#GP fault
Virtual-8086, VME ³	3	IF = 0
	0-2	VIF = 0

NOTES:

1. X = This setting has no effect on instruction operation.

2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.

3. PVI mode and virtual-8086 mode each imply CPL = 3.

Operation

```

IF CR0.PE = 0
  THEN IF := 0; (* Reset Interrupt Flag *)
  ELSE
    IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
      THEN IF := 0; (* Reset Interrupt Flag *)
      ELSE
        IF VME mode OR PVI mode
          THEN VIF := 0; (* Reset Virtual Interrupt Flag *)
          ELSE #GP(0);
        FI;
      FI;
    FI;
  FI;

```

Flags Affected

Either the IF flag or the VIF flag is cleared to 0. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active. If CPL is greater than IOPL and less than 3.
#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 IOPL is less than 3 and VME mode is not active.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 CLRSSBSY m64	M	V/V	CET_SS	Clear busy flag in supervisor shadow stack token reference by m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
M	N/A	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Clear busy flag in supervisor shadow stack token reference by m64. Subsequent to marking the shadow stack as not busy the SSP is loaded with value 0.

Operation

IF (CR4.CET = 0)
THEN #UD; FI;

IF (IA32_S_CET.SH_STK_EN = 0)
THEN #UD; FI;

IF CPL > 0
THEN GP(0); FI;

SSP_LA = Linear_Address(mem operand)

IF SSP_LA not aligned to 8 bytes

THEN #GP(0); FI;

expected_token_value = SSP_LA | BUSY_BIT (* busy bit - bit position 0 - must be set *)

new_token_value = SSP_LA (* Clear the busy bit *)

IF shadow_stack_lock_cmpxchg8b(SSP_LA, new_token_value, expected_token_value) != expected_token_value

invalid_token := 1; FI

(* Set the CF if invalid token was detected *)

RFLAGS.CF = (invalid_token == 1) ? 1 : 0;

RFLAGS.ZF,PF,AF,OF,SF := 0;

SSP := 0

Flags Affected

CF is set if an invalid token was detected, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

Protected Mode Exceptions

#UD	<p>If the LOCK prefix is used.</p> <p>If CR4.CET = 0.</p> <p>IF IA32_S_CET.SH_STK_EN = 0.</p>
#GP(0)	<p>If memory operand linear address not aligned to 8 bytes.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If destination is located in a non-writeable segment.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If CPL is not 0.</p>
#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	The CLRSSBSY instruction is not recognized in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

#UD	The CLRSSBSY instruction is not recognized in virtual-8086 mode.
-----	--

Compatibility Mode Exceptions

#UD	Same exceptions as in protected mode.
#GP(0)	Same exceptions as in protected mode.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	<p>If the LOCK prefix is used.</p> <p>If CR4.CET = 0.</p> <p>IF IA32_S_CET.SH_STK_EN = 0.</p>
#GP(0)	<p>If memory operand linear address not aligned to 8 bytes.</p> <p>If CPL is not 0.</p> <p>If the memory address is in a non-canonical form.</p> <p>If token is invalid.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF 06	CLTS	Z0	Valid	Valid	Clears TS flag in CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 26, “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

CR0.TS[bit 3] := 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.

CLUI—Clear User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EE CLUI	Z0	V/I	UINTR	Clear user interrupt flag; user interrupts blocked when user interrupt flag cleared.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A	N/A

Description

CLUI clears the user interrupt flag (UIF). Its effect takes place immediately: a user interrupt cannot be delivered on the instruction boundary following CLUI.

An execution of CLUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been caused due to an execution of CLI.

Operation

UIF := 0;

Flags Affected

None.

Protected Mode Exceptions

#UD The CLUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The CLUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The CLUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The CLUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD

- If the LOCK prefix is used.
- If executed inside an enclave.
- If CR4.UINTR = 0.
- If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.

CLWB—Cache Line Write Back

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 CLWB m8	M	V/V	CLWB	Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state.

Instruction Operand Encoding¹

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

Description

Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

The availability of CLWB instruction is indicated by the presence of the CPUID feature flag CLWB (bit 24 of the EBX 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 that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHH 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 CLWB instruction is not ordered with respect to PREFETCHH 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 CLWB instruction that references the cache line).

Executions of the CLWB instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to older writes to the cache line being written back. They are not ordered with respect to other executions of CLWB, to executions of CLFLUSH and CLFLUSHOPT, or to younger writes to the cache line being written back. Software can use the SFENCE instruction to order an execution of CLWB relative to one of those operations.

For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the accessed flag but not the dirty flag in the page tables.

In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

Operation

Cache_Line_Write_Back(m8);

1. The Mod field of the ModR/M byte cannot have value 11B.

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

```
CLWB void _mm_clwb(void const *p);
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#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.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.

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

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#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.

CMC—Complement Carry Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
F5	CMC	Z0	Valid	Valid	Complement CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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
OF 47 /r	CMOVA r16, r/m16	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
OF 47 /r	CMOVA r32, r/m32	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVA r64, r/m64	RM	Valid	N.E.	Move if above (CF=0 and ZF=0).
OF 43 /r	CMOVAE r16, r/m16	RM	Valid	Valid	Move if above or equal (CF=0).
OF 43 /r	CMOVAE r32, r/m32	RM	Valid	Valid	Move if above or equal (CF=0).
REX.W + OF 43 /r	CMOVAE r64, r/m64	RM	Valid	N.E.	Move if above or equal (CF=0).
OF 42 /r	CMOVB r16, r/m16	RM	Valid	Valid	Move if below (CF=1).
OF 42 /r	CMOVB r32, r/m32	RM	Valid	Valid	Move if below (CF=1).
REX.W + OF 42 /r	CMOVB r64, r/m64	RM	Valid	N.E.	Move if below (CF=1).
OF 46 /r	CMOVBE r16, r/m16	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
OF 46 /r	CMOVBE r32, r/m32	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + OF 46 /r	CMOVBE r64, r/m64	RM	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).
OF 42 /r	CMOVC r16, r/m16	RM	Valid	Valid	Move if carry (CF=1).
OF 42 /r	CMOVC r32, r/m32	RM	Valid	Valid	Move if carry (CF=1).
REX.W + OF 42 /r	CMOVC r64, r/m64	RM	Valid	N.E.	Move if carry (CF=1).
OF 44 /r	CMOVE r16, r/m16	RM	Valid	Valid	Move if equal (ZF=1).
OF 44 /r	CMOVE r32, r/m32	RM	Valid	Valid	Move if equal (ZF=1).
REX.W + OF 44 /r	CMOVE r64, r/m64	RM	Valid	N.E.	Move if equal (ZF=1).
OF 4F /r	CMOVG r16, r/m16	RM	Valid	Valid	Move if greater (ZF=0 and SF=0F).
OF 4F /r	CMOVG r32, r/m32	RM	Valid	Valid	Move if greater (ZF=0 and SF=0F).
REX.W + OF 4F /r	CMOVG r64, r/m64	RM	V/N.E.	N/A	Move if greater (ZF=0 and SF=0F).
OF 4D /r	CMOVGE r16, r/m16	RM	Valid	Valid	Move if greater or equal (SF=0F).
OF 4D /r	CMOVGE r32, r/m32	RM	Valid	Valid	Move if greater or equal (SF=0F).
REX.W + OF 4D /r	CMOVGE r64, r/m64	RM	Valid	N.E.	Move if greater or equal (SF=0F).
OF 4C /r	CMOVL r16, r/m16	RM	Valid	Valid	Move if less (SF≠ 0F).
OF 4C /r	CMOVL r32, r/m32	RM	Valid	Valid	Move if less (SF≠ 0F).
REX.W + OF 4C /r	CMOVL r64, r/m64	RM	Valid	N.E.	Move if less (SF≠ 0F).
OF 4E /r	CMOVLE r16, r/m16	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ 0F).
OF 4E /r	CMOVLE r32, r/m32	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ 0F).
REX.W + OF 4E /r	CMOVLE r64, r/m64	RM	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ 0F).
OF 46 /r	CMOVNA r16, r/m16	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
OF 46 /r	CMOVNA r32, r/m32	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + OF 46 /r	CMOVNA r64, r/m64	RM	Valid	N.E.	Move if not above (CF=1 or ZF=1).
OF 42 /r	CMOVNAE r16, r/m16	RM	Valid	Valid	Move if not above or equal (CF=1).
OF 42 /r	CMOVNAE r32, r/m32	RM	Valid	Valid	Move if not above or equal (CF=1).
REX.W + OF 42 /r	CMOVNAE r64, r/m64	RM	Valid	N.E.	Move if not above or equal (CF=1).
OF 43 /r	CMOVNB r16, r/m16	RM	Valid	Valid	Move if not below (CF=0).
OF 43 /r	CMOVNB r32, r/m32	RM	Valid	Valid	Move if not below (CF=0).
REX.W + OF 43 /r	CMOVNB r64, r/m64	RM	Valid	N.E.	Move if not below (CF=0).
OF 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	CMOVPP r16, r/m16	RM	Valid	Valid	Move if parity (PF=1).
OF 4A /r	CMOVPP r32, r/m32	RM	Valid	Valid	Move if parity (PF=1).
REX.W + OF 4A /r	CMOVPP 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	CMOVVS r16, r/m16	RM	Valid	Valid	Move if sign (SF=1).
OF 48 /r	CMOVVS r32, r/m32	RM	Valid	Valid	Move if sign (SF=1).
REX.W + OF 48 /r	CMOVVS 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)	N/A	N/A

Description

Each of the CMOVcc instructions performs a move operation if the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) 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.

Specifically, CMOVcc loads data from its source operand into a temporary register unconditionally (regardless of the condition code and the status flags in the EFLAGS register). If the condition code associated with the instruction (cc) is satisfied, the data in the temporary register is then copied into the instruction's destination operand.

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 0F 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;

ELSE IF (OperandSize = 32 and IA-32e mode active)

THEN DEST[63:32] := 0;

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.

CMP—Compare Two Operands

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

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	N/A	N/A
MR	ModRM:r/m (r)	ModRM:reg (r)	N/A	N/A
MI	ModRM:r/m (r)	imm8/16/32	N/A	N/A
I	AL/AX/EAX/RAX (r)	imm8/16/32	N/A	N/A

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 Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.66.0F.WIG C2 /r ib VCMPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.66.0F.WIG C2 /r ib VCMPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.66.0F.W1 C2 /r ib VCMPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed double precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.66.0F.W1 C2 /r ib VCMPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed double precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.66.0F.W1 C2 /r ib VCMPD k1 {k2}, zmm2, zmm3/m512/m64bcst{sae}, imm8	C	V/V	AVX512F	Compare packed double precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

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 result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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 (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions

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	
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, nonsignaling)	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
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No

Table 3-1. Comparison Predicate for CMPPD and CMPPS 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	
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	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.

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 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 the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) 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 3-2. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-2. Pseudo-Op and CMPPD Implementation

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

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.)

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-3, 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 3-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The 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. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-3.

Table 3-3. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMPPEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0</i>
VCMPPLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1</i>
VCMPLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 3</i>
VCMPNEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 4</i>
VCMPNLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 5</i>
VCMPNLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 6</i>
VCMPORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 7</i>
VCMPPEQ_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 8</i>
VCMPNGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 9</i>
VCMPNGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0CH</i>
VCMPGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0DH</i>
VCMPGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0FH</i>
VCMPPEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 10H</i>
VCMPPLT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 12H</i>
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>
VCMPPEQ_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>
VCMPTRUE_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1FH</i>

Operation

CASE (COMPARISON PREDICATE) OF

```

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

VCMPDPD (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k2[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

CMP := SRC1[i+63:i] OP5 SRC2[63:0]

ELSE

CMP := SRC1[i+63:i] OP5 SRC2[i+63:i]

FI;

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] := 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX_KL-1:KL] := 0

VCMPDPD (VEX.256 Encoded Version)

CMP0 := SRC1[63:0] OP5 SRC2[63:0];

CMP1 := SRC1[127:64] OP5 SRC2[127:64];

CMP2 := SRC1[191:128] OP5 SRC2[191:128];

CMP3 := SRC1[255:192] OP5 SRC2[255:192];

IF CMP0 = TRUE

THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;

ELSE DEST[63:0] := 0000000000000000H; FI;

IF CMP1 = TRUE

THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;

ELSE DEST[127:64] := 0000000000000000H; FI;

IF CMP2 = TRUE

THEN DEST[191:128] := FFFFFFFFFFFFFFFFH;

ELSE DEST[191:128] := 0000000000000000H; FI;

IF CMP3 = TRUE

THEN DEST[255:192] := FFFFFFFFFFFFFFFFH;

ELSE DEST[255:192] := 0000000000000000H; FI;

DEST[MAXVL-1:256] := 0

VCMPDPD (VEX.128 Encoded Version)

CMP0 := SRC1[63:0] OP5 SRC2[63:0];

CMP1 := SRC1[127:64] OP5 SRC2[127:64];

IF CMP0 = TRUE

THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;

ELSE DEST[63:0] := 0000000000000000H; FI;

IF CMP1 = TRUE

THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;

ELSE DEST[127:64] := 0000000000000000H; FI;

DEST[MAXVL-1:128] := 0

CMPPD (128-bit Legacy SSE Version)

```

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

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCMPD __mmask8 __mm512_cmp_pd_mask( __m512d a, __m512d b, int imm);
VCMPD __mmask8 __mm512_cmp_round_pd_mask( __m512d a, __m512d b, int imm, int sae);
VCMPD __mmask8 __mm512_mask_cmp_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm);
VCMPD __mmask8 __mm512_mask_cmp_round_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPD __mmask8 __mm256_cmp_pd_mask( __m256d a, __m256d b, int imm);
VCMPD __mmask8 __mm256_mask_cmp_pd_mask( __mmask8 k1, __m256d a, __m256d b, int imm);
VCMPD __mmask8 __mm_cmp_pd_mask( __m128d a, __m128d b, int imm);
VCMPD __mmask8 __mm_mask_cmp_pd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPD __m256 __mm256_cmp_pd(__m256d a, __m256d b, int imm)
(V)VCMPD __m128 __mm_cmp_pd(__m128d a, __m128d b, int imm)

```

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

CMPPS—Compare Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.128.OF.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.256.OF.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.128.OF.W0 C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed single precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.256.OF.W0 C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed single precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.512.OF.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	C	V/V	AVX512F	Compare packed single precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

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 result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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 (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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 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 the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*) 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 3-4. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-4. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 0
CMPLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 1
CMPLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 2
CMPUNORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 3
CMPNEQPS xmm1, xmm2	CMPPS xmm1, xmm2, 4
CMPNLTPS xmm1, xmm2	CMPPS xmm1, xmm2, 5
CMPNLEPS xmm1, xmm2	CMPPS xmm1, xmm2, 6
CMPORDPS xmm1, xmm2	CMPPS xmm1, xmm2, 7

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.)

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-5, 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 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. The 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. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.

Table 3-5. 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>
VCMPUNORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 3</i>
VCMPNEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 4</i>
VCMPNLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 5</i>
VCMPNLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 6</i>
VCMPORDPS <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>
VCMPNGEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 9</i>
VCMPNGTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0CH</i>
VCMPGEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0DH</i>
VCMPGTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 11H</i>
VCMPL_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 12H</i>
VCMPUNORD_SPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1EH</i>
VCMPTRUE_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1FH</i>

Operation

```

CASE (COMPARISON PREDICATE) OF
  0: OP3 := EQ_OQ; OP5 := EQ_OQ;
  1: OP3 := LT_OS; OP5 := LT_OS;
  2: OP3 := LE_OS; OP5 := LE_OS;
  3: OP3 := UNORD_Q; OP5 := UNORD_Q;
  4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
  5: OP3 := NLT_US; OP5 := NLT_US;
  6: OP3 := NLE_US; OP5 := NLE_US;
  7: OP3 := ORD_Q; OP5 := ORD_Q;
  8: OP5 := EQ_UQ;
  9: OP5 := NGE_US;
 10: OP5 := NGT_US;
 11: OP5 := FALSE_OQ;
 12: OP5 := NEQ_OQ;
 13: OP5 := GE_OS;
 14: OP5 := GT_OS;
 15: OP5 := TRUE_UQ;
 16: OP5 := EQ_OS;
 17: OP5 := LT_OQ;
 18: OP5 := LE_OQ;
 19: OP5 := UNORD_S;
 20: OP5 := 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
ESAC;

```

VCMPPS (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

i := j * 32

IF k2[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

CMP := SRC1[i+31:i] OP5 SRC2[31:0]

ELSE

CMP := SRC1[i+31:i] OP5 SRC2[i+31:i]

FI;

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] := 0 ; zeroing-masking only FI;

FI;

ENDFOR

DEST[MAX_KL-1:KL] := 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;

DEST[MAXVL-1:256] := 0

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[MAXVL-1:128] := 0

```

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[MAXVL-1:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCMPPS __mmask16 __mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPPS __mmask16 __mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 __mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 __mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask8 __mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPPS __mmask8 __mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 __mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPPS __mmask8 __mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 __mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 __mm_cmp_ps(__m128 a, __m128 b, int imm)

```

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

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>	Z0	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)ESI to byte at address (R)EDI. The status flags are set accordingly.
A7	CMPS <i>m16, m16</i>	Z0	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)ESI with word at address (R)EDI. The status flags are set accordingly.
A7	CMPS <i>m32, m32</i>	Z0	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)ESI at dword at address (R)EDI. The status flags are set accordingly.
REX.W + A7	CMPS <i>m64, m64</i>	Z0	Valid	N.E.	Compares quadword at address (R)ESI with quadword at address (R)EDI and sets the status flags accordingly.
A6	CMPSB	Z0	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)ESI with byte at address (R)EDI. The status flags are set accordingly.
A7	CMPSW	Z0	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)ESI with word at address (R)EDI. The status flags are set accordingly.
A7	CMPSD	Z0	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)ESI with dword at address (R)EDI. The status flags are set accordingly.
REX.W + A7	CMPSQ	Z0	Valid	N.E.	Compares quadword at address (R)ESI with quadword at address (R)EDI and sets the status flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 loca-

tion. 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;
```

```
        FI;
```

```

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;
    FI;
FI;
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;
            FI;
        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;
            FI;
        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;
            FI;
        FI;
    FI;
FI;

```

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 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F2.0F.WIG C2 /r ib VCMPD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F2.0F.W1 C2 /r ib VCMPD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	C	V/V	AVX512F	Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

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. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

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 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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 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 the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*) 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 3-6. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-6. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 0
CMPLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 1
CMPLESD xmm1, xmm2	CMPSD xmm1, xmm2, 2
CMPUNORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 3
CMPNEQSD xmm1, xmm2	CMPSD xmm1, xmm2, 4
CMPNLTSD xmm1, xmm2	CMPSD xmm1, xmm2, 5
CMPNLESD xmm1, xmm2	CMPSD xmm1, xmm2, 6
CMPORDSD xmm1, xmm2	CMPSD xmm1, xmm2, 7

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.)

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-7, 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 3-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The 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. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	VCMPSD Implementation
VCMPSEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0
VCMPLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1
VCMPLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 2
VCMPUNORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 3
VCMPNEQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 4
VCMPNLTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 5
VCMPNLESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 6
VCMPORDSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 7
VCMPSEQ_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 8
VCMPNGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 9
VCMPNGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0AH
VCMPFALSESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0BH
VCMPNEQ_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0CH
VCMPGESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0DH

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	VCMPSD Implementation
VCMPGTSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0EH
VCMPTRUESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0FH
VCMPSEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 10H
VCMPLE_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
VCMPSEQ_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
VCMPTTRUE_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1FH

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

CASE (COMPARISON PREDICATE) OF

```

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

```

VCMPSD (EVEX Encoded Version)

```
CMPO := SRC1[63:0] OP5 SRC2[63:0];
```

```

IF k2[0] or *no writemask*
    THEN    IF CMPO = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0

```

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[MAXVL-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[MAXVL-1:128] := 0

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCMPSD __mmask8 __mm_cmp_sd_mask( __m128d a, __m128d b, int imm);
VCMPSD __mmask8 __mm_cmp_round_sd_mask( __m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 __mm_mask_cmp_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 __mm_mask_cmp_round_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d __mm_cmp_sd(__m128d a, __m128d b, const int imm)

```

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

CMPSS—Compare Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.LLIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	C	V/V	AVX512F	Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

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 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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. The result is stored in the low 32 bits of the destination operand; bits 127:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

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 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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 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” 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 the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*) 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 3-8. The compiler should treat reserved imm8 values as illegal syntax.

Table 3-8. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 0
CMPLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 1
CMPLESS xmm1, xmm2	CMPSS xmm1, xmm2, 2
CMPUNORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 3
CMPNEQSS xmm1, xmm2	CMPSS xmm1, xmm2, 4
CMPNLTSS xmm1, xmm2	CMPSS xmm1, xmm2, 5
CMPNLESS xmm1, xmm2	CMPSS xmm1, xmm2, 6
CMPORDSS xmm1, xmm2	CMPSS xmm1, xmm2, 7

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.)

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPS instruction. See Table 3-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The 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. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPS instructions in a similar fashion by extending the syntax listed in Table 3-9.

Table 3-9. Pseudo-Op and VCMPS Implementation

Pseudo-Op	VCMPS Implementation
VCMPEQSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 0
VCMPLTSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 1
VCMPLESS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 2
VCMPUNORDSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 3
VCMPNEQSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 4
VCMPNLTSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 5
VCMPNLESS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 6
VCMPORDSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 7
VCMPEQ_UQSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 8
VCMPNGESS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 9
VCMPNGTSS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 0AH
VCMPPALSESS reg1, reg2, reg3	VCMPS reg1, reg2, reg3, 0BH

Table 3-9. Pseudo-Op and VCMPPSS Implementation

Pseudo-Op	CMPPSS Implementation
VCMPPNEQ_OQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 0CH
VCMPPGESS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 0DH
VCMPPGTSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 0EH
VCMPPTRUESS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 0FH
VCMPEQ_OSSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 10H
VCMPLT_OQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 11H
VCMPLT_OQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 12H
VCMPPUNORD_SSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 13H
VCMPPNEQ_USSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 14H
VCMPPNLT_UQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 15H
VCMPPNLE_UQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 16H
VCMPPORD_SSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 17H
VCMPEQ_USSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 18H
VCMPPNGE_UQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 19H
VCMPPNGT_UQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 1AH
VCMPPFALSE_OSSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 1BH
VCMPPNEQ_OSSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 1CH
VCMPPGE_OQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 1DH
VCMPPGT_OQSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 1EH
VCMPPTRUE_USSS reg1, reg2, reg3	VCMPPSS reg1, reg2, reg3, 1FH

Software should ensure VCMPPSS is encoded with VEX.L=0. Encoding VCMPPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

CASE (COMPARISON PREDICATE) OF

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

```

19: OP5 := UNORD_S;
20: OP5 := 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

```

ESAC;

VCMPSS (EVEX Encoded Version)

CMPO := SRC1[31:0] OP5 SRC2[31:0];

```

IF k2[0] or *no writemask*
    THEN    IF CMPO = TRUE
                THEN DEST[0] := 1;
                ELSE DEST[0] := 0; FI;
    ELSE    DEST[0] := 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0

```

CMPS (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[MAXVL-1:32] (Unmodified)

```

VCMPSS (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[MAXVL-1:128] := 0

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCMPSS __mmask8 __mm_cmp_ss_mask( __m128 a, __m128 b, int imm);
VCMPSS __mmask8 __mm_cmp_round_ss_mask( __m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 __mm_mask_cmp_ss_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 __mm_mask_cmp_round_ss_mask( __mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __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 Table 3-1, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

CMPXCHG—Compare and Exchange

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

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;

FI;

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 CMPXCHG8B 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.
REX.W + OF C7 /1 CMPXCHG16B 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.

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)	N/A	N/A	N/A

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 3-242.

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;
  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.
#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 Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LLIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F	Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

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).

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 operation exception only if a source operand is an SNaN.

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

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISD (All Versions)

```

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 Equivalent

```

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd (__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comile_sd (__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comige_sd (__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd (__m128d a, __m128d b)

```

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LLIG.OF.WO 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F	Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Compares the single 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).

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 operation exception only if a source operand is an SNaN.

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

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISS (All Versions)

```

RESULT := OrderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF := 111;
  GREATER_THAN: ZF,PF,CF := 000;
  LESS_THAN: ZF,PF,CF := 001;
  EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCOMISS int __mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int __mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int __mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int __mm_comile_ss (__m128 a, __m128 b)
VCOMISS int __mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int __mm_comige_ss (__m128 a, __m128 b)
VCOMISS int __mm_comineq_ss (__m128 a, __m128 b)

```

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

Additionally:

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
-----	---

CPUID—CPU Identification

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F A2	CPUID	Z0	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
Z0	N/A	N/A	N/A	N/A

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 3-8 shows information returned, depending on the initial value loaded into the EAX register.

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 some Intel processors, 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. *)2
CPUID.EAX = 1FH (* Returns V2 Extended Topology Enumeration leaf. *)2
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.

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

"Caching Translation Information" in Chapter 4, "Paging," 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.

2. CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of CPUID leaf 1FH before using leaf 0BH.

Table 3-8. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
Basic CPUID Information		
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information. "Genu" "ntel" "inel"
01H	EAX EBX ECX EDX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6). Bits 07-00: Brand Index. Bits 15-08: CLFLUSH line size (Value * 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID**. Feature Information (see Figure 3-7 and Table 3-10). Feature Information (see Figure 3-8 and Table 3-11). 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. ** The 8-bit initial APIC ID in EBX[31:24] is replaced by the 32-bit x2APIC ID, available in Leaf 0BH and Leaf 1FH.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-12). Cache and TLB Information. Cache and TLB Information. Cache and TLB Information.
03H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00-31 of 96-bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) 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.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
Deterministic Cache Parameters Leaf		
04H	EAX	NOTES: Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 250. Bits 04-00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

Table 3-8. 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.</p> <p>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 00: 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 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: 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> <p>* If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0.</p> <p>** Add one to the return value to get the result.</p> <p>***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.</p> <p>**** 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.</p> <p>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p>	
	<i>MONITOR/MWAIT Leaf</i>	
05H	<p>EAX Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity). Bits 31-16: Reserved = 0.</p> <p>EBX Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity). Bits 31-16: Reserved = 0.</p> <p>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.</p> <p>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.</p>	

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	NOTE: * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.	
Thermal and Power Management Leaf		
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_ENABLE[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. Bit 14: Intel® Turbo Boost Max Technology 3.0 available. Bit 15: HWP Capabilities. Highest Performance change is supported if set. Bit 16: HWP PECL override is supported if set. Bit 17: Flexible HWP is supported if set. Bit 18: Fast access mode for the IA32_HWP_REQUEST MSR is supported if set. Bit 19: HW_FEEDBACK. IA32_HW_FEEDBACK_PTR MSR, IA32_HW_FEEDBACK_CONFIG MSR, IA32_PACKAGE_THERM_STATUS MSR bit 26, and IA32_PACKAGE_THERM_INTERRUPT MSR bit 25 are supported if set. Bit 20: Ignoring Idle Logical Processor HWP request is supported if set. Bits 22-21: Reserved. Bit 23: Intel® Thread Director supported if set. IA32_HW_FEEDBACK_CHAR and IA32_HW_FEEDBACK_THREAD_CONFIG MSRs are supported if set. Bit 24: IA32_THERM_INTERRUPT MSR bit 25 is supported if set. Bits 31-25: Reserved.
	EBX	Bits 03-00: Number of Interrupt Thresholds in Digital Thermal Sensor. Bits 31-04: Reserved.
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. 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 (1BOH). Bits 07-04: Reserved = 0. Bits 15-08: Number of Intel® Thread Director classes supported by the processor. Information for that many classes is written into the Intel Thread Director Table by the hardware. Bits 31-16: Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>EDX Bits 07-00: Bitmap of supported hardware feedback interface capabilities. 0 = When set to 1, indicates support for performance capability reporting. 1 = When set to 1, indicates support for energy efficiency capability reporting. 2-7 = Reserved</p> <p> Bits 11-08: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages; add one to the return value to get the result. Bits 31-16: Index (starting at 0) of this logical processor's row in the hardware feedback interface structure. Note that on some parts the index may be same for multiple logical processors. On some parts the indices may not be contiguous, i.e., there may be unused rows in the hardware feedback interface structure.</p> <p>NOTE: Bits 0 and 1 will always be set together.</p>
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>	
07H	<p> Sub-leaf 0 (Input ECX = 0). *</p> <p>EAX Bits 31-00: Reports the maximum input value for supported leaf 7 sub-leaves.</p> <p>EBX Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1. Bit 03: BMI1. Bit 04: HLE. Bit 05: AVX2. Supports Intel® Advanced Vector Extensions 2 (Intel® AVX2) if 1. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2. Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM. Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1. Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. Bit 16: AVX512F. Bit 17: AVX512DQ. Bit 18: RDSEED. Bit 19: ADX. Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bit 21: AVX512_IFMA. Bit 22: Reserved. Bit 23: CLFLUSHOPT. Bit 24: CLWB. Bit 25: Intel Processor Trace. Bit 26: AVX512PF. (Intel® Xeon Phi™ only.) Bit 27: AVX512ER. (Intel® Xeon Phi™ only.) Bit 28: AVX512CD. Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1. Bit 30: AVX512BW. Bit 31: AVX512VL.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
ECX	<p>Bit 00: PREFETCHWT1. (Intel® Xeon Phi™ only.)</p> <p>Bit 01: AVX512_VBMI.</p> <p>Bit 02: UMIP. Supports user-mode instruction prevention if 1.</p> <p>Bit 03: PKU. Supports protection keys for user-mode pages if 1.</p> <p>Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions).</p> <p>Bit 05: WAITPKG.</p> <p>Bit 06: AVX512_VBMI2.</p> <p>Bit 07: CET_SS. Supports CET shadow stack features if 1. Processors that set this bit define bits 1:0 of the IA32_U_CET and IA32_S_CET MSRs. Enumerates support for the following MSRs: IA32_INTERRUPT_SPP_TABLE_ADDR, IA32_PL3_SSP, IA32_PL2_SSP, IA32_PL1_SSP, and IA32_PL0_SSP.</p> <p>Bit 08: GFNI.</p> <p>Bit 09: VAES.</p> <p>Bit 10: VPCLMULQDQ.</p> <p>Bit 11: AVX512_VNNI.</p> <p>Bit 12: AVX512_BITALG.</p> <p>Bits 13: TME_EN. If 1, the following MSRs are supported: IA32_TME_CAPABILITY, IA32_TME_ACTIVATE, IA32_TME_EXCLUDE_MASK, and IA32_TME_EXCLUDE_BASE.</p> <p>Bit 14: AVX512_VPOPCNTDQ.</p> <p>Bit 15: Reserved.</p> <p>Bit 16: LA57. Supports 57-bit linear addresses and five-level paging if 1.</p> <p>Bits 21-17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode.</p> <p>Bit 22: RDPID and IA32_TSC_AUX are available if 1.</p> <p>Bit 23: KL. Supports Key Locker if 1.</p> <p>Bit 24: BUS_LOCK_DETECT. If 1, indicates support for OS bus-lock detection.</p> <p>Bit 25: CLDEMOTE. Supports cache line demote if 1.</p> <p>Bit 26: Reserved.</p> <p>Bit 27: MOVDIRI. Supports MOVDIRI if 1.</p> <p>Bit 28: MOVDIR64B. Supports MOVDIR64B if 1.</p> <p>Bit 29: ENQCMD. Supports Enqueue Stores if 1.</p> <p>Bit 30: SGX_LC. Supports SGX Launch Configuration if 1.</p> <p>Bit 31: PKS. Supports protection keys for supervisor-mode pages if 1.</p>
EDX	<p>Bit 00: Reserved.</p> <p>Bit 01: SGX-KEYS. If 1, Attestation Services for Intel® SGX is supported.</p> <p>Bit 02: AVX512_4VNNIW. (Intel® Xeon Phi™ only.)</p> <p>Bit 03: AVX512_4FMAPS. (Intel® Xeon Phi™ only.)</p> <p>Bit 04: Fast Short REP MOV.</p> <p>Bit 05: UINTR. If 1, the processor supports user interrupts.</p> <p>Bits 07-06: Reserved.</p> <p>Bit 08: AVX512_VP2INTERSECT.</p> <p>Bit 09: SRBDS_CTRL. If 1, enumerates support for the IA32_MCU_OPT_CTRL MSR and indicates its bit 0 (RNGDS_MITG_DIS) is also supported.</p> <p>Bit 10: MD_CLEAR supported.</p> <p>Bit 11: RTM_ALWAYS_ABORT. If set, any execution of XBEGIN immediately aborts and transitions to the specified fallback address.</p> <p>Bit 12: Reserved.</p> <p>Bit 13: If 1, RTM_FORCE_ABORT supported. Processors that set this bit support the IA32_TSX_FORCE_ABORT MSR. They allow software to set IA32_TSX_FORCE_ABORT[0] (RTM_FORCE_ABORT).</p> <p>Bit 14: SERIALIZE.</p> <p>Bit 15: Hybrid. If 1, the processor is identified as a hybrid part. If CPUID.0.MAXLEAF ≥ 1AH and CPUID.1A.EAX ≠ 0, then the Native Model ID Enumeration Leaf 1AH exists.</p> <p>Bit 16: TSXLDTRK. If 1, the processor supports Intel TSX suspend/resume of load address tracking.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	<p>Bit 17: Reserved.</p> <p>Bit 18: PCONFIG. Supports PCONFIG if 1.</p> <p>Bit 19: Architectural LBRs. If 1, indicates support for architectural LBRs.</p> <p>Bit 20: CET_IBT. Supports CET indirect branch tracking features if 1. Processors that set this bit define bits 5:2 and bits 63:10 of the IA32_U_CET and IA32_S_CET MSRs.</p> <p>Bit 21: Reserved.</p> <p>Bit 22: AMX-BF16. If 1, the processor supports tile computational operations on bfloat16 numbers.</p> <p>Bit 23: AVX512_FP16.</p> <p>Bit 24: AMX-TILE. If 1, the processor supports tile architecture.</p> <p>Bits 25: AMX-INT8. If 1, the processor supports tile computational operations on 8-bit integers.</p> <p>Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB).</p> <p>Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP).</p> <p>Bit 28: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH).</p> <p>Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR.</p> <p>Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR.</p> <p>IA32_CORE_CAPABILITIES is an architectural MSR that enumerates model-specific features. A bit being set in this MSR indicates that a model specific feature is supported; software must still consult CPUID family/model/stepping to determine the behavior of the enumerated feature as features enumerated in IA32_CORE_CAPABILITIES may have different behavior on different processor models. Some of these features may have behavior that is consistent across processor models (and for which consultation of CPUID family/model/stepping is not necessary); such features are identified explicitly where they are documented in this manual.</p> <p>Bit 31: Enumerates support for Speculative Store Bypass Disable (SSBD). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[2] (SSBD).</p> <p>NOTE:</p> <p>* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.</p>
<i>Structured Extended Feature Enumeration Sub-leaf (EAX = 07H, ECX = 1)</i>	
07H	<p>NOTES:</p> <p>Leaf 07H output depends on the initial value in ECX.</p> <p>If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.</p> <p>EAX</p> <p>This field reports 0 if the sub-leaf index, 1, is invalid.</p> <p>Bits 03-00: Reserved.</p> <p>Bit 04: AVX-VNNI. AVX (VEX-encoded) versions of the Vector Neural Network Instructions.</p> <p>Bit 05: AVX512_BF16. Vector Neural Network Instructions supporting BFLOAT16 inputs and conversion instructions from IEEE single precision.</p> <p>Bits 09-06: Reserved.</p> <p>Bit 10: If 1, supports fast zero-length REP MOVSB.</p> <p>Bit 11: If 1, supports fast short REP STOSB.</p> <p>Bit 12: If 1, supports fast short REP CMPSB, REP SCASB.</p> <p>Bits 21-13: Reserved.</p> <p>Bit 22: HRESET. If 1, supports history reset via the HRESET instruction and the IA32_HRESET_ENABLE MSR. When set, indicates that the Processor History Reset Leaf (EAX = 20H) is valid.</p> <p>Bits 31-23: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	<p>This field reports 0 if the sub-leaf index, 1, is invalid.</p> <p>Bit 00: Enumerates the presence of the IA32_PPIN and IA32_PPIN_CTL MSRs. If 1, these MSRs are supported.</p> <p>Bits 31-01: Reserved.</p>
	ECX	This field reports 0 if the sub-leaf index, 1, is invalid; otherwise it is reserved.
	EDX	<p>This field reports 0 if the sub-leaf index, 1, is invalid.</p> <p>Bits 17-00: Reserved.</p> <p>Bit 18: CET_SSS. If 1, indicates that an operating system can enable supervisor shadow stacks as long as it ensures that certain supervisor shadow-stack pushes will not cause page faults (see Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). When emulating the CPUID instruction, a virtual-machine monitor should return this bit as 0 if those pushes can cause VM exits.</p> <p>Bits 31-19: Reserved.</p>
<i>Structured Extended Feature Enumeration Sub-leaf (EAX = 07H, ECX = 2)</i>		
07H		<p>NOTES:</p> <p>Leaf 07H output depends on the initial value in ECX.</p> <p>If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.</p>
	EAX	This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.
	ECX	This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.
	EDX	<p>This field reports 0 if the sub-leaf index, 2, is invalid.</p> <p>Bit 00: PSFD. If 1, indicates bit 7 of the IA32_SPEC_CTRL MSR is supported. Bit 7 of this MSR disables Fast Store Forwarding Predictor without disabling Speculative Store Bypass.</p> <p>Bit 01: IPRED_CTRL. If 1, indicates bits 3 and 4 of the IA32_SPEC_CTRL MSR are supported. Bit 3 of this MSR enables IPRED_DIS control for CPL3. Bit 4 of this MSR enables IPRED_DIS control for CPL0/1/2.</p> <p>Bit 02: RRSBA_CTRL. If 1, indicates bits 5 and 6 of the IA32_SPEC_CTRL MSR are supported. Bit 5 of this MSR disables RRSBA behavior for CPL3. Bit 6 of this MSR disables RRSBA behavior for CPL0/1/2.</p> <p>Bit 03: DDPD_U. If 1, indicates bit 8 of the IA32_SPEC_CTRL MSR is supported. Bit 8 of this MSR disables Data Dependent Prefetcher.</p> <p>Bit 04: BHI_CTRL. If 1, indicates bit 10 of the IA32_SPEC_CTRL MSR is supported. Bit 10 of this MSR enables BHI_DIS_S behavior.</p> <p>Bit 05: MCDT_NO. Processors that enumerate this bit as 1 do not exhibit MXCSR Configuration Dependent Timing (MCDT) behavior and do not need to be mitigated to avoid data-dependent behavior for certain instructions.</p> <p>Bits 31-06: 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. Architectural event x is supported if EBX[x]=0 && EAX[31:24]>x.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: Core cycle event not available if 1 or if EAX[31:24]<1. Bit 01: Instruction retired event not available if 1 or if EAX[31:24]<2. Bit 02: Reference cycles event not available if 1 or if EAX[31:24]<3. Bit 03: Last-level cache reference event not available if 1 or if EAX[31:24]<4. Bit 04: Last-level cache misses event not available if 1 or if EAX[31:24]<5. Bit 05: Branch instruction retired event not available if 1 or if EAX[31:24]<6. Bit 06: Branch mispredict retired event not available if 1 or if EAX[31:24]<7. Bit 07: Top-down slots event not available if 1 or if EAX[31:24]<8. Bits 31-08: Reserved = 0.
	ECX	Bits 31-00: Supported fixed counters bit mask. Fixed-function performance counter 'i' is supported if bit 'i' is 1 (first counter index starts at zero). It is recommended to use the following logic to determine if a Fixed Counter is supported: $FxCtr[i]_{is_supported} := ECX[i] \parallel (EDX[4:0] > i)$;
	EDX	Bits 04-00: Number of contiguous fixed-function performance counters starting from 0 (if Version ID > 1). Bits 12-05: Bit width of fixed-function performance counters (if Version ID > 1). Bits 14-13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31-16: Reserved = 0.
<i>Extended Topology Enumeration Leaf</i>		
OBH	<p>NOTES:</p> <p><i>CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.</i></p> <p>The sub-leaves of CPUID leaf 0BH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket).</p> <p>The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated.</p> <p>For two valid sub-leaves N and N+1, sub-leaf N+1 represents the next immediate higher-scoped domain with respect to the domain of sub-leaf N for the given processor.</p> <p>If sub-leaf index "N" returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than "N" shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.</p>	
	EAX	Bits 04-00: The number of bits that the x2APIC ID must be shifted to the right to address instances of the next higher-scoped domain. When logical processor is not supported by the processor, the value of this field at the Logical Processor domain sub-leaf may be returned as either 0 (no allocated bits in the x2APIC ID) or 1 (one allocated bit in the x2APIC ID); software should plan accordingly. Bits 31-05: Reserved.
	EBX	Bits 15-00: The number of logical processors across all instances of this domain within the next higher-scoped domain. (For example, in a processor socket/package comprising "M" dies of "N" cores each, where each core has "L" logical processors, the "die" domain sub-leaf value of this field would be M*N*L.) This number reflects configuration as shipped by Intel. Note, software must not use this field to enumerate processor topology*. Bits 31-16: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor										
	ECX	<p>Bits 07-00: The input ECX sub-leaf index.</p> <p>Bits 15-08: Domain Type. This field provides an identification value which indicates the domain as shown below. Although domains are ordered, their assigned identification values are not and software should not depend on it.</p> <table> <tr> <th><u>Hierarchy</u></th><th><u>Domain</u></th><th><u>Domain Type Identification Value</u></th></tr> <tr> <td>Lowest</td><td>Logical Processor</td><td>1</td></tr> <tr> <td>Highest</td><td>Core</td><td>2</td></tr> </table> <p>(Note that enumeration values of 0 and 3-255 are reserved.)</p> <p>Bits 31-16: Reserved.</p>	<u>Hierarchy</u>	<u>Domain</u>	<u>Domain Type Identification Value</u>	Lowest	Logical Processor	1	Highest	Core	2
<u>Hierarchy</u>	<u>Domain</u>	<u>Domain Type Identification Value</u>									
Lowest	Logical Processor	1									
Highest	Core	2									
	EDX	<p>Bits 31-00: x2APIC ID of the current logical processor.</p> <p>NOTES:</p> <p>* Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.</p>									
<i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>											
0DH		<p>NOTES:</p> <p>Leaf 0DH main leaf (ECX = 0).</p>									
	EAX	<p>Bits 31-00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1.</p> <p>Bit 00: x87 state.</p> <p>Bit 01: SSE state.</p> <p>Bit 02: AVX state.</p> <p>Bits 04-03: MPX state.</p> <p>Bits 07-05: AVX-512 state.</p> <p>Bit 08: Used for IA32_XSS.</p> <p>Bit 09: PKRU state.</p> <p>Bits 16-10: Used for IA32_XSS.</p> <p>Bit 17: TILECFG state.</p> <p>Bit 18: TILEDATA state.</p> <p>Bits 31-19: Reserved.</p>									
	EBX	<p>Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. 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 XCR0.</p>									
	EDX	<p>Bit 31-00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1.</p> <p>Bits 31-00: Reserved.</p>									
<i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>											
0DH	EAX	<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> <p>Bit 04: Supports extended feature disable (XFD) if set.</p> <p>Bits 31-05: Reserved.</p>									

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCRO IA32_XSS. NOTES: If EAX[3] is enumerated as 0 and EAX[1] is enumerated as 1, EBX enumerates the size of the XSAVE area containing all states enabled by XCRO. If EAX[1] and EAX[3] are both enumerated as 0, EBX enumerates zero.
	ECX	Bits 31-00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07-00: Used for XCRO. Bit 08: PT state. Bit 09: Used for XCRO. Bit 10: PASID state. Bit 11: CET user state. Bit 12: CET supervisor state. Bit 13: HDC state. Bit 14: UINTR state. Bit 15: LBR state (only for the architectural LBR feature). Bit 16: HWP state. Bits 18-17: Used for XCRO. Bits 31-19: Reserved.
	EDX	Bits 31-00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31-00: Reserved.
<i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)</i>		
0DH		NOTES: Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCRO register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n ($0 \leq n \leq 31$) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n ($32 \leq n \leq 63$) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31-00: 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.
	EBX	Bits 31-00: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCRO register*.
	ECX	Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCRO. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31-02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*.
	EDX	This field reports 0 if the sub-leaf index, n, is invalid*; otherwise it is reserved.
<i>Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)</i>		
0FH		NOTES: Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 31-00: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31-02: Reserved.
<i>L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)</i>		
0FH		NOTES: Leaf 0FH output depends on the initial value in ECX.
	EAX	Reserved.
	EBX	Bits 31-00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes) and Memory Bandwidth Monitoring (MBM) metrics.
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31-03: Reserved.
<i>Intel Resource Director Technology (Intel RDT) Allocation 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 EBX.
	EAX	Reserved.
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31-04: Reserved.
	ECX	Reserved.
	EDX	Reserved.
<i>L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 1)</i>		
10H		NOTES: Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04-00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31-05: Reserved.
	EBX	Bits 31-00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 01-00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31-03: Reserved.
	EDX	Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)		
10H	NOTES: Leaf 10H output depends on the initial value in ECX.	
EAX	Bits 04-00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31-05: Reserved.	
EBX	Bits 31-00: Bit-granular map of isolation/contention of allocation units.	
ECX	Bits 31-00: Reserved.	
EDX	Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved.	
Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)		
10H	NOTES: Leaf 10H output depends on the initial value in ECX.	
EAX	Bits 11-00: Reports the maximum MBA throttling value supported for the corresponding ResID. Add one to the return value to get the result. Bits 31-12: Reserved.	
EBX	Bits 31-00: Reserved.	
ECX	Bits 01-00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31-03: Reserved.	
EDX	Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved.	
Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)		
12H	NOTES: Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.	
EAX	Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bits 04-02: Reserved. Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVMCHILD, EDECVIRCHILD, and ESETCONTEXT. Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC. Bit 07: If 1, indicates Intel SGX supports ENCLU instruction leaf EVERIFYREPORT2. Bits 09-08: Reserved. Bit 10: If 1, indicates Intel SGX supports ENCLS instruction leaf EUPDATESVN. Bit 11: If 1, indicates Intel SGX supports ENCLU instruction leaf EDECCSSA. Bits 31-12: Reserved.	
EBX	Bits 31-00: MISCSELECT. Bit vector of supported extended SGX features.	
ECX	Bits 31-00: Reserved.	
EDX	Bits 07-00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is 2^(EDX[7:0]). Bits 15-08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is 2^(EDX[15:8]). Bits 31-16: Reserved.	

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
<i>Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)</i>	
12H	<p>NOTES: Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.</p> <p>EAX Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.</p> <p>EBX Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.</p> <p>ECX Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.</p> <p>EDX Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.</p>
<i>Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)</i>	
12H	<p>NOTES: Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.</p> <p>EAX Bit 03-00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.</p> <p>Type 0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.</p> <p>Type 0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If ECX[3:0] = 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If ECX[3:0] = 0001b, then this section has confidentiality and integrity protection. If ECX[3:0] = 0010b, then this section has confidentiality protection only. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.</p>
<i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>	
14H	<p>NOTES: Leaf 14H main leaf (ECX = 0).</p> <p>EAX Bits 31-00: Reports the maximum sub-leaf supported in leaf 14H.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	<p>Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed.</p> <p>Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode.</p> <p>Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset.</p> <p>Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets.</p> <p>Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets.</p> <p>Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation.</p> <p>Bit 06: If 1, indicates support for PSB and PMI preservation. Writes can set IA32_RTIT_CTL[56] (InjectPsb-PmiOnEnable), enabling the processor to set IA32_RTIT_STATUS[7] (PendTopaPMI) and/or IA32_RTIT_STATUS[6] (PendPSB) in order to preserve ToPA PMIs and/or PSBs otherwise lost due to Intel PT disable. Writes can also set PendToPAPMI and PendPSB.</p> <p>Bit 07: If 1, writes can set IA32_RTIT_CTL[31] (EventEn), enabling Event Trace packet generation.</p> <p>Bit 08: If 1, writes can set IA32_RTIT_CTL[55] (DistNT), disabling TNT packet generation.</p> <p>Bit 31-09: Reserved.</p>
	ECX	<p>Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed.</p> <p>Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.</p> <p>Bit 02: If 1, indicates support of Single-Range Output scheme.</p> <p>Bit 03: If 1, indicates support of output to Trace Transport subsystem.</p> <p>Bit 30-04: Reserved.</p> <p>Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.</p>
	EDX	Bits 31-00: Reserved.
<i>Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)</i>		
14H	EAX	<p>Bits 02-00: Number of configurable Address Ranges for filtering.</p> <p>Bits 15-03: Reserved.</p> <p>Bits 31-16: Bitmap of supported MTC period encodings.</p>
	EBX	<p>Bits 15-00: Bitmap of supported Cycle Threshold value encodings.</p> <p>Bit 31-16: Bitmap of supported Configurable PSB frequency encodings.</p>
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
<i>Time Stamp Counter and Nominal Core Crystal Clock Information Leaf</i>		
15H	<p>NOTES:</p> <p>If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated.</p> <p>EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency.</p> <p>If ECX is 0, the nominal core crystal clock frequency is not enumerated.</p> <p>"TSC frequency" = "core crystal clock frequency" * EBX/EAX.</p> <p>The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</p>	
	EAX	Bits 31-00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31-00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31-00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31-00: Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
Processor Frequency Information Leaf		
16H	EAX	Bits 15-00: Processor Base Frequency (in MHz). Bits 31-16: Reserved =0.
	EBX	Bits 15-00: Maximum Frequency (in MHz). Bits 31-16: Reserved = 0.
	ECX	Bits 15-00: Bus (Reference) Frequency (in MHz). Bits 31-16: Reserved = 0.
	EDX	Reserved.
	NOTES: * Data is returned from this interface in accordance with the processor’s specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces. While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.	
System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)		
17H	NOTES: Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index >= 3. Leaf 17H sub-leaves 4 and above are reserved.	
	EAX	Bits 31-00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.
	EBX	Bits 15-00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31-17: Reserved = 0.
	ECX	Bits 31-00: Project ID. A unique number an SOC vendor assigns to its SOC projects.
	EDX	Bits 31-00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.
System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)		
17H	EAX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.
	NOTES: Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.	

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)</i>	
17H	<p>NOTES:</p> <p>Leaf 17H output depends on the initial value in ECX.</p> <p>EAX Bits 31-00: Reserved = 0.</p> <p>EBX Bits 31-00: Reserved = 0.</p> <p>ECX Bits 31-00: Reserved = 0.</p> <p>EDX Bits 31-00: Reserved = 0.</p>
<i>Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)</i>	
18H	<p>NOTES:</p> <p>Each sub-leaf enumerates a different address translation structure.</p> <p>If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.</p> <p>* Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product.</p> <p>** Add one to the return value to get the result.</p> <p>EAX Bits 31-00: Reports the maximum input value of supported sub-leaf in leaf 18H.</p> <p>EBX Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07-04: Reserved. Bits 10-08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15-11: Reserved. Bits 31-16: W = Ways of associativity.</p> <p>ECX Bits 31-00: S = Number of Sets.</p> <p>EDX Bits 04-00: Translation cache type field. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: Unified TLB*. 00100b: Load Only TLB. Hit on loads; fills on both loads and stores. 00101b: Store Only TLB. Hit on stores; fill on stores. All other encodings are reserved. Bits 07-05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13-09: Reserved. Bits 25-14: Maximum number of addressable IDs for logical processors sharing this translation cache.** Bits 31-26: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)		
18H	NOTES:	<p>Each sub-leaf enumerates a different address translation structure.</p> <p>If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.</p> <p>* Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch. See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product.</p> <p>** Add one to the return value to get the result.</p>
	EAX	Bits 31-00: Reserved.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07-04: Reserved. Bits 10-08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15-11: Reserved. Bits 31-16: W = Ways of associativity.
	ECX	Bits 31-00: S = Number of Sets.
	EDX	Bits 04-00: Translation cache type field. 0000b: Null (indicates this sub-leaf is not valid). 0001b: Data TLB. 0010b: Instruction TLB. 0011b: Unified TLB*. All other encodings are reserved. Bits 07-05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13-09: Reserved. Bits 25-14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31-26: Reserved.
Key Locker Leaf (EAX = 19H)		
19H	EAX	Bit 00: Key Locker restriction of CPL0-only supported. Bit 01: Key Locker restriction of no-encrypt supported. Bit 02: Key Locker restriction of no-decrypt supported. Bits 31-03: Reserved.
	EBX	Bit 00: AESKLE. If 1, the AES Key Locker instructions are fully enabled. Bit 01: Reserved. Bit 02: If 1, the AES wide Key Locker instructions are supported. Bit 03: Reserved. Bit 04: If 1, the platform supports the Key Locker MSRs (IA32_COPY_LOCAL_TO_PLATFORM, IA32_COPY_PLATFORM_TO_LOCAL, IA32_COPY_STATUS, and IA32_IWKEYBACKUP_STATUS) and backing up the internal wrapping key. Bits 31-05: Reserved.
	ECX	Bit 00: If 1, the NoBackup parameter to LOADIWKEY is supported. Bit 01: If 1, KeySource encoding of 1 (randomization of the internal wrapping key) is supported. Bits 31-02: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
	EDX Reserved.
<i>Native Model ID Enumeration Leaf (EAX = 1AH, ECX = 0)</i>	
1AH	<p>NOTES:</p> <p>This leaf exists on all hybrid parts, however this leaf is not only available on hybrid parts. The following algorithm is used for detection of this leaf: If CPUID.0.MAXLEAF \geq 1AH and CPUID.1A.EAX \neq 0, then the leaf exists.</p> <p>EAX Enumerates the native model ID and core type. Bits 31-24: Core type* 10H: Reserved 20H: Intel Atom® 30H: Reserved 40H: Intel® Core™ Bits 23-00: Native model ID of the core. The core-type and native model ID can be used to uniquely identify the microarchitecture of the core. This native model ID is not unique across core types, and not related to the model ID reported in CPUID leaf 01H, and does not identify the SOC. * The core type may only be used as an identification of the microarchitecture for this logical processor and its numeric value has no significance, neither large nor small. This field neither implies nor expresses any other attribute to this logical processor and software should not assume any.</p> <p>EBX Reserved.</p> <p>ECX Reserved.</p> <p>EDX Reserved.</p>
<i>PCONFIG Information Sub-leaf (EAX = 1BH, ECX \geq 0)</i>	
1BH	<p>For details on this sub-leaf, see "INPUT EAX = 1BH: Returns PCONFIG Information" on page 3-252.</p> <p>NOTE:</p> <p>Leaf 1BH is supported if CPUID.(EAX=07H, ECX=0H):EDX[18] = 1.</p>
<i>Last Branch Records Information Leaf (EAX = 1CH)</i>	
1CH	<p>NOTE:</p> <p>This leaf pertains to the architectural feature.</p> <p>EAX Bits 07-00: Supported LBR Depth Values. For each bit n set in this field, the IA32_LBR_DEPTH.DEPTH value $8*(n+1)$ is supported. Bits 29-08: Reserved. Bit 30: Deep C-state Reset. If set, indicates that LBRs may be cleared on an MWAIT that requests a C-state numerically greater than C1. Bit 31: IP Values Contain LIP. If set, LBR IP values contain LIP. If clear, IP values contain Effective IP.</p> <p>EBX Bit 00: CPL Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[2:1] to non-zero value. Bit 01: Branch Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[22:16] to non-zero value. Bit 02: Call-stack Mode Supported. If set, the processor supports setting IA32_LBR_CTL[3] to 1. Bits 31-03: Reserved.</p> <p>ECX Bit 00: Mispredict Bit Supported. IA32_LBR_x_INFO[63] holds indication of branch misprediction (MISPRED). Bit 01: Timed LBRs Supported. IA32_LBR_x_INFO[15:0] holds CPU cycles since last LBR entry (CYC_CNT), and IA32_LBR_x_INFO[60] holds an indication of whether the value held there is valid (CYC_CNT_VALID). Bit 02: Branch Type Field Supported. IA32_LBR_INFO_x[59:56] holds indication of the recorded operation's branch type (BR_TYPE). Bits 31-03: Reserved.</p>

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 31-00: Reserved.
<i>Tile Information Main Leaf (EAX = 1DH, ECX = 0)</i>		
1DH	NOTES: For sub-leaves of 1DH, they are indexed by the palette id. Leaf 1DH sub-leaves 2 and above are reserved.	
	EAX	Bits 31-00: max_palette. Highest numbered palette sub-leaf. Value = 1.
	EBX	Bits 31-00: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
<i>Tile Palette 1 Sub-leaf (EAX = 1DH, ECX = 1)</i>		
1DH	EAX	Bits 15-00: Palette 1 total_tile_bytes. Value = 8192. Bits 31-16: Palette 1 bytes_per_tile. Value = 1024.
	EBX	Bits 15-00: Palette 1 bytes_per_row. Value = 64. Bits 31-16: Palette 1 max_names (number of tile registers). Value = 8.
	ECX	Bits 15-00: Palette 1 max_rows. Value = 16. Bits 31-16: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
<i>TMUL Information Main Leaf (EAX = 1EH, ECX = 0)</i>		
1EH	NOTE: Leaf 1EH sub-leaves 1 and above are reserved.	
	EAX	Bits 31-00: Reserved = 0.
	EBX	Bits 07-00: tmul_maxk (rows or columns). Value = 16. Bits 23-08: tmul_maxn (column bytes). Value = 64. Bits 31-24: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
<i>V2 Extended Topology Enumeration Leaf</i>		
1FH	NOTES: <i>CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends using leaf 1FH when available rather than leaf 0BH and ensuring that any leaf 0BH algorithms are updated to support leaf 1FH.</i> The sub-leaves of CPUID leaf 1FH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket). The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated. As an example, a processor may report an ordered hierarchy consisting only of "Logical Processor," "Core," and "Die." For two valid sub-leaves N and N+1, sub-leaf N+1 represents the next immediate higher-scoped domain with respect to the domain of sub-leaf N for the given processor. If sub-leaf index "N" returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than "N" shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.	

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor																									
	EAX	<p>Bits 04-00: The number of bits that the x2APIC ID must be shifted to the right to address instances of the next higher-scoped domain. When logical processor is not supported by the processor, the value of this field at the Logical Processor domain sub-leaf may be returned as either 0 (no allocated bits in the x2APIC ID) or 1 (one allocated bit in the x2APIC ID); software should plan accordingly.</p> <p>Bits 31-05: Reserved.</p>																								
	EBX	<p>Bits 15-00: The number of logical processors across all instances of this domain within the next higher-scoped domain. (For example, in a processor socket/package comprising “M” dies of “N” cores each, where each core has “L” logical processors, the “die” domain sub-leaf value of this field would be M*N*L.) This number reflects configuration as shipped by Intel. Note, software must not use this field to enumerate processor topology*.</p> <p>Bits 31-16: Reserved.</p>																								
	ECX	<p>Bits 07-00: The input ECX sub-leaf index.</p> <p>Bits 15-08: Domain Type. This field provides an identification value which indicates the domain as shown below. Although domains are ordered, as also shown below, their assigned identification values are not and software should not depend on it. (For example, if a new domain between core and module is specified, it will have an identification value higher than 5.)</p> <table> <tr> <th><u>Hierarchy</u></th><th><u>Domain</u></th><th><u>Domain Type Identification Value</u></th></tr> <tr> <td>Lowest</td><td>Logical Processor</td><td>1</td></tr> <tr> <td>...</td><td>Core</td><td>2</td></tr> <tr> <td>...</td><td>Module</td><td>3</td></tr> <tr> <td>...</td><td>Tile</td><td>4</td></tr> <tr> <td>...</td><td>Die</td><td>5</td></tr> <tr> <td>...</td><td>DieGrp</td><td>6</td></tr> <tr> <td>Highest</td><td>Package/Socket</td><td>(implied)</td></tr> </table> <p>(Note that enumeration values of 0 and 7-255 are reserved.)</p> <p>Bits 31-16: Reserved.</p>	<u>Hierarchy</u>	<u>Domain</u>	<u>Domain Type Identification Value</u>	Lowest	Logical Processor	1	...	Core	2	...	Module	3	...	Tile	4	...	Die	5	...	DieGrp	6	Highest	Package/Socket	(implied)
<u>Hierarchy</u>	<u>Domain</u>	<u>Domain Type Identification Value</u>																								
Lowest	Logical Processor	1																								
...	Core	2																								
...	Module	3																								
...	Tile	4																								
...	Die	5																								
...	DieGrp	6																								
Highest	Package/Socket	(implied)																								
	EDX	<p>Bits 31-00: x2APIC ID of the current logical processor. It is always valid and does not vary with the sub-leaf index in ECX.</p> <p>NOTES:</p> <p>* Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.</p>																								
<i>Processor History Reset Sub-leaf (EAX = 20H, ECX = 0)</i>																										
20H	EAX	Reports the maximum number of sub-leaves that are supported in leaf 20H.																								
	EBX	<p>Indicates which bits may be set in the IA32_HRESET_ENABLE MSR to enable reset of different components of hardware-maintained history.</p> <p>Bit 00: Indicates support for both HRESET's EAX[0] parameter, and IA32_HRESET_ENABLE[0] set by the OS to enable reset of Intel® Thread Director history.</p> <p>Bits 31-01: Reserved = 0.</p>																								
	ECX	Reserved.																								
	EDX	Reserved.																								
<i>Unimplemented CPUID Leaf Functions</i>																										
21H		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is 21H. If the value returned by CPUID.0:EAX (the maximum input value for basic CPUID information) is at least 21H, 0 is returned in the registers EAX, EBX, ECX, and EDX. Otherwise, the data for the highest basic information leaf is returned.																								

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
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.	
Extended Function CPUID Information		
80000000H	EAX EBX ECX EDX	Maximum Input Value for Extended Function CPUID Information. Reserved. Reserved. Reserved.
80000001H	EAX EBX ECX <	

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
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.
	EDX	Reserved = 0.
	NOTES:	
	* L2 associativity field encodings:	
	00H - Disabled	08H - 16 ways
	01H - 1 way (direct mapped)	09H - Reserved
	02H - 2 ways	0AH - 32 ways
	03H - Reserved	0BH - 48 ways
04H - 4 ways	0CH - 64 ways	
05H - Reserved	0DH - 96 ways	
06H - 8 ways	0EH - 128 ways	
07H - See CPUID leaf 04H, sub-leaf 2**	0FH - Fully associative	
** CPUID leaf 04H provides details of deterministic cache parameters, including the L2 cache in sub-leaf 2		
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 15-08: #Linear Address Bits. Bits 31-16: Reserved = 0.
	EBX	Bits 08-00: Reserved = 0. Bit 09: WBNOINVD is available if 1. Bits 31-10: 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 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 and is processor specific.

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 10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). 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 3-9 for available processor type values. Stepping IDs are provided as needed.

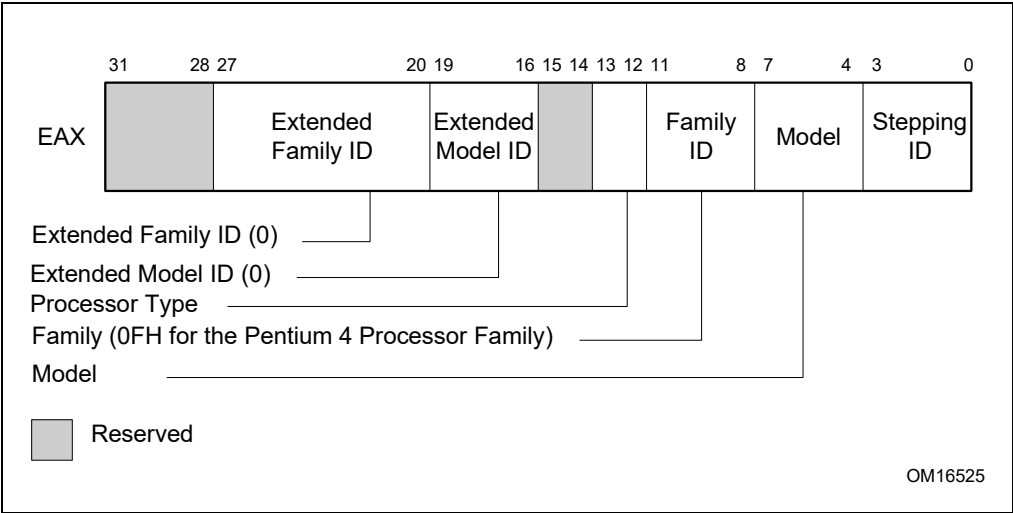


Figure 3-6. Version Information Returned by CPUID in EAX

Table 3-9. 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 20 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;
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 = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, 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 by the CLFLUSH and CLFLUSHOPT instructions 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 = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 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.

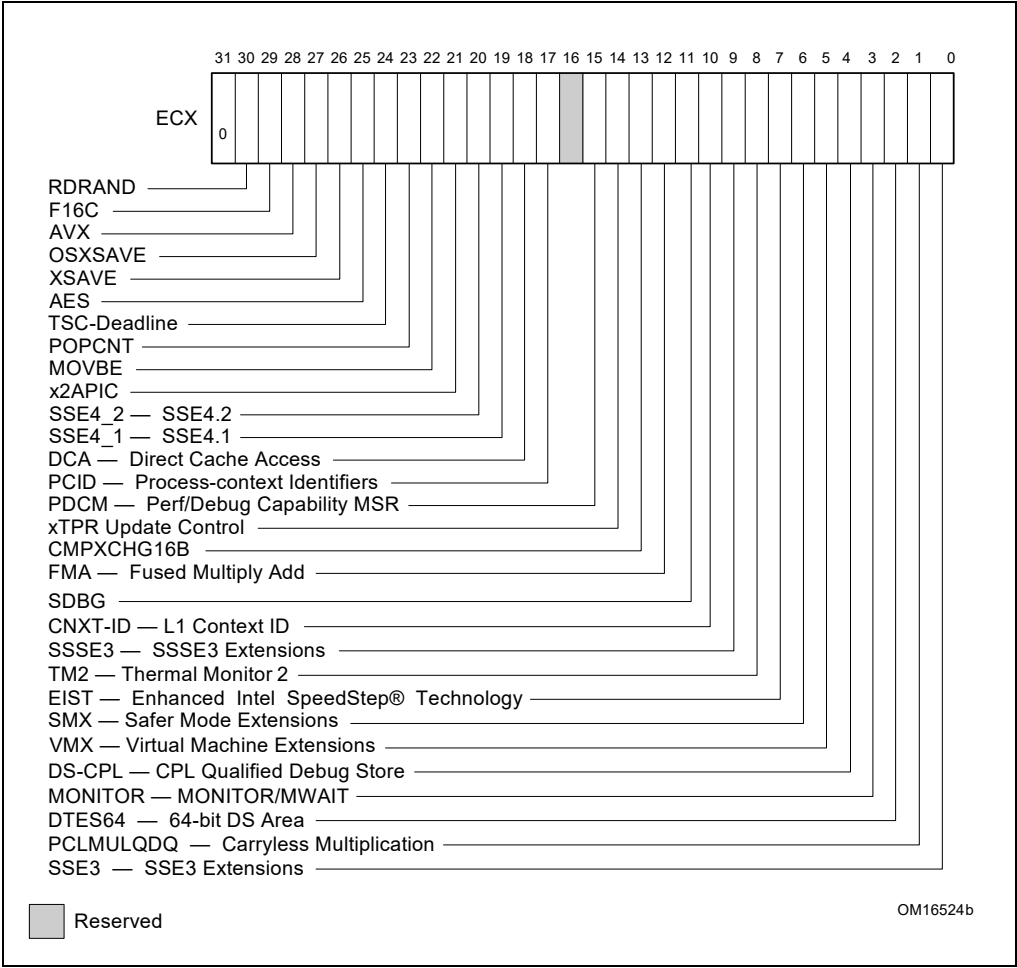


Figure 3-7. Feature Information Returned in the ECX Register

Table 3-10. 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 7, “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 3-10. 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 XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
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.

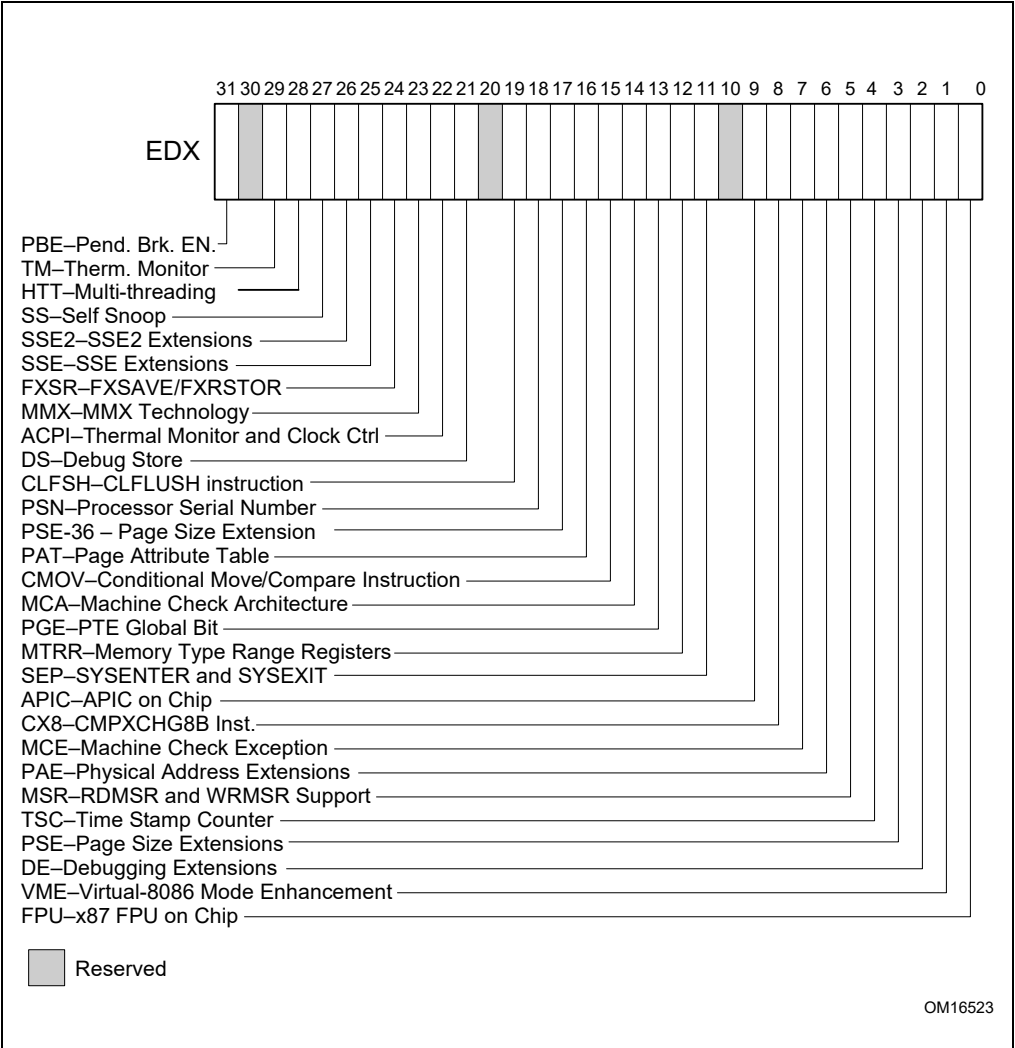


Figure 3-8. Feature Information Returned in the EDX Register

Table 3-11. 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. A value of 1 indicates the Machine Check Architecture of reporting machine errors 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 3-11. 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 processor event-based sampling (PEBS) facilities (see Chapter 24, “Introduction to Virtual-Machine Extensions,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual</i> , Volume 3C).
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.

INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, 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) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- 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 3-12. Table 3-12 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 3-12. 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 3-12. 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 TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 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: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 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
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
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

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
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
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
C0H	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
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GByte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 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
FEH	General	CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.
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 value should be ignored.
- 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 sector, 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 3-8.

This Cache Size in Bytes

$$= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line_Size} + 1) * (\text{Sets} + 1)$$

$$= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

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 9, “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 3-8.

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 3-8.

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 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), 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 3-8.

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 3-8) is greater than Pn 0. See Table 3-8.

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 24, “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

CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.

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 3-8.

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 3-8.

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 3-8. 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;

FI;

INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) 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 3-8.

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 Intel Resource Director Technology (Intel RDT) Allocation 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 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 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.

INPUT EAX = 12H: Returns Intel SGX Enumeration Information

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

INPUT EAX = 17H: Returns System-On-Chip Information

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

INPUT EAX = 19H: Returns Key Locker Information

When CPUID executes with EAX set to 19H, the processor returns information about Key Locker. See Table 3-8.

INPUT EAX = 1AH: Returns Native Model ID Information

When CPUID executes with EAX set to 1AH, the processor returns information about Native Model Identification. See Table 3-8.

INPUT EAX = 1BH: Returns PCONFIG Information

When CPUID executes with EAX set to 1BH, the processor returns information about PCONFIG capabilities. This information is enumerated in sub-leaves selected by the value of ECX (starting with 0).

Each sub-leaf of CPUID function 1BH enumerates its **sub-leaf type** in EAX. If a sub-leaf type is 0, the sub-leaf is invalid and zero is returned in EBX, ECX, and EDX. In this case, all subsequent sub-leaves (selected by larger input values of ECX) are also invalid.

The only valid sub-leaf type currently defined is 1, indicating that the sub-leaf enumerates target identifiers for the PCONFIG instruction. Any non-zero value returned in EBX, ECX, or EDX indicates a valid target identifier of the PCONFIG instruction (any value of zero should be ignored). The only target identifier currently defined is 1, indicating MKTME. See the “PCONFIG—Platform Configuration” instruction in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B for more information.

INPUT EAX = 1CH: Returns Last Branch Record Information

When CPUID executes with EAX set to 1CH, the processor returns information about LBRs (the architectural feature). See Table 3-8.

INPUT EAX = 1DH: Returns Tile Information

When CPUID executes with EAX set to 1DH and ECX = 0H, the processor returns information about tile architecture. See Table 3-8.

When CPUID executes with EAX set to 1DH and ECX = 1H, the processor returns information about tile palette 1. See Table 3-8.

INPUT EAX = 1EH: Returns TMUL Information

When CPUID executes with EAX set to 1EH and ECX = 0H, the processor returns information about TMUL capabilities. See Table 3-8.

INPUT EAX = 1FH: Returns V2 Extended Topology Information

When CPUID executes with EAX set to 1FH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 1FH by verifying (a) the highest leaf index supported by CPUID is $\geq 1FH$, and (b) CPUID.1FH:EBX[15:0] reports a non-zero value. See Table 3-8.

INPUT EAX = 20H: Returns History Reset Information

When CPUID executes with EAX set to 20H, the processor returns information about History Reset. See Table 3-8.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method.
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 20 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The Processor Brand String Method

Figure 3-9 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 Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.

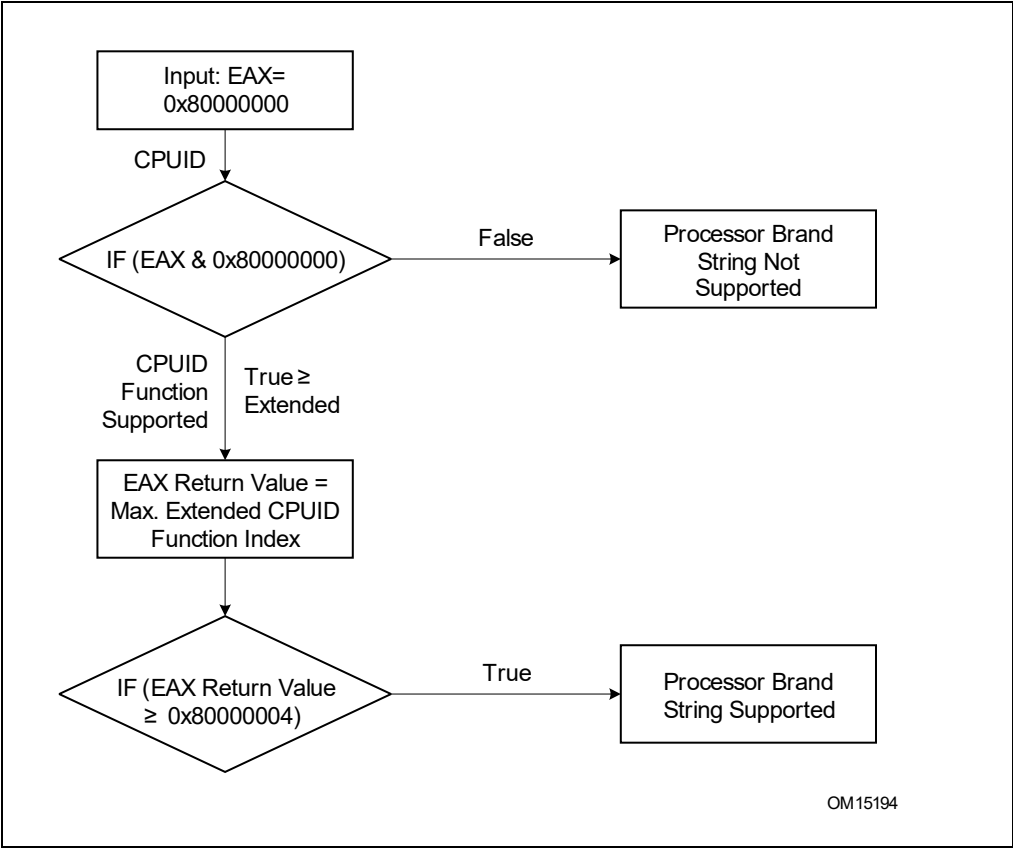


Figure 3-9. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-13. Processor Brand String Returned with Pentium 4 Processor

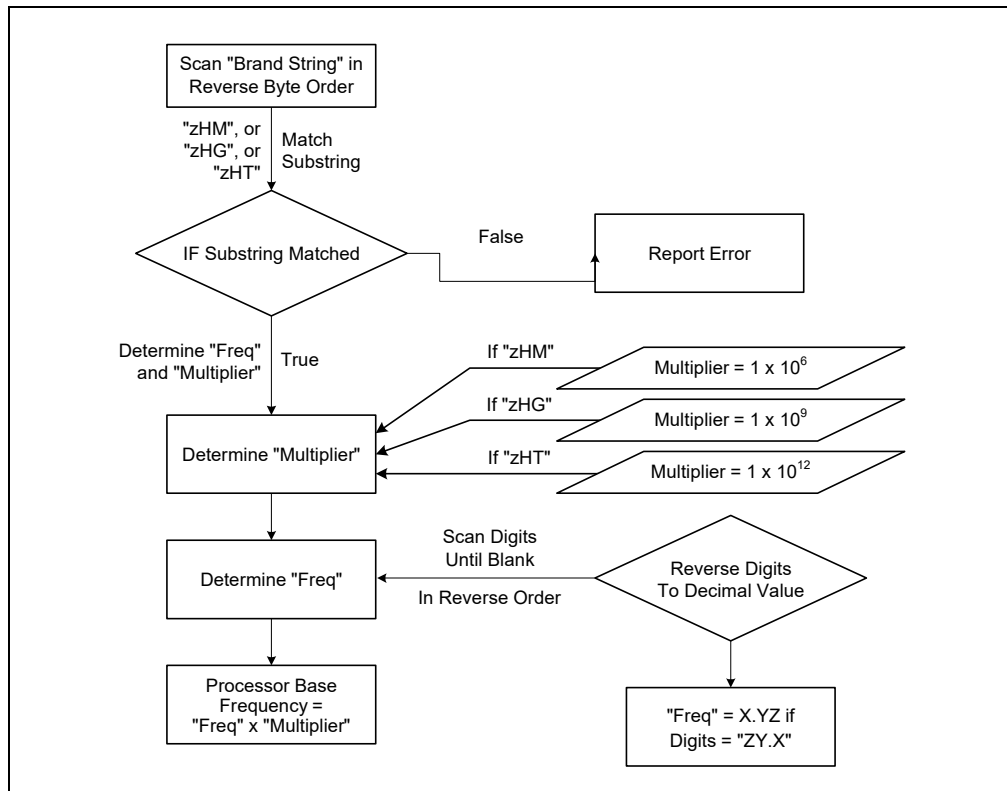
EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	" " " " " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P)R" "itne" "R(mu"

Table 3-13. Processor Brand String Returned with Pentium 4 Processor (Contd.)

EAX Input Value	Return Values	ASCII Equivalent
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4)" " UPC" "0051" "\0zHM"

Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

**Figure 3-10. Algorithm for Extracting 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 3-14 shows brand indices that have identification strings associated with them.

Table 3-14. 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
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 – 0FFH	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 3-7. *)
EDX := Feature flags; (* See Figure 3-8. *)
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 3-8. *)
    EBX := Deterministic Cache Parameters Leaf;
    ECX := Deterministic Cache Parameters Leaf;
    EDX := Deterministic Cache Parameters Leaf;
BREAK;
EAX = 5H:
    EAX := MONITOR/MWAIT Leaf; (* See Table 3-8. *)
    EBX := MONITOR/MWAIT Leaf;
    ECX := MONITOR/MWAIT Leaf;
    EDX := MONITOR/MWAIT Leaf;
BREAK;
EAX = 6H:
    EAX := Thermal and Power Management Leaf; (* See Table 3-8. *)
    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 3-8. *)
    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 3-8. *)
    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 3-8. *)
    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 3-8. *)
    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 3-8. *)
    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 := Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    ECX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    EDX := Intel Resource Director Technology Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
    EAX := Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel Resource Director Technology Allocation Enumeration Leaf;
    ECX := Intel Resource Director Technology Allocation Enumeration Leaf;
    EDX := Intel Resource Director Technology Allocation Enumeration Leaf;
BREAK;
EAX = 12H:
    EAX := Intel SGX Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel SGX Enumeration Leaf;
    ECX := Intel SGX Enumeration Leaf;

```


EDX := Intel SGX Enumeration Leaf;
 BREAK;
 EAX = 14H:
 EAX := Intel Processor Trace Enumeration Leaf; (* See Table 3-8. *)
 EBX := Intel Processor Trace Enumeration Leaf;
 ECX := Intel Processor Trace Enumeration Leaf;
 EDX := Intel Processor Trace Enumeration Leaf;
 BREAK;
 EAX = 15H:
 EAX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (* See Table 3-8. *)
 EBX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
 ECX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
 EDX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
 BREAK;
 EAX = 16H:
 EAX := Processor Frequency Information Enumeration Leaf; (* See Table 3-8. *)
 EBX := Processor Frequency Information Enumeration Leaf;
 ECX := Processor Frequency Information Enumeration Leaf;
 EDX := Processor Frequency Information Enumeration Leaf;
 BREAK;
 EAX = 17H:
 EAX := System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 3-8. *)
 EBX := System-On-Chip Vendor Attribute Enumeration Leaf;
 ECX := System-On-Chip Vendor Attribute Enumeration Leaf;
 EDX := System-On-Chip Vendor Attribute Enumeration Leaf;
 BREAK;
 EAX = 18H:
 EAX := Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 3-8. *)
 EBX := Deterministic Address Translation Parameters Enumeration Leaf;
 ECX := Deterministic Address Translation Parameters Enumeration Leaf;
 EDX := Deterministic Address Translation Parameters Enumeration Leaf;
 BREAK;
 EAX = 19H:
 EAX := Key Locker Enumeration Leaf; (* See Table 3-8. *)
 EBX := Key Locker Enumeration Leaf;
 ECX := Key Locker Enumeration Leaf;
 EDX := Key Locker Enumeration Leaf;
 BREAK;
 EAX = 1AH:
 EAX := Native Model ID Enumeration Leaf; (* See Table 3-8. *)
 EBX := Native Model ID Enumeration Leaf;
 ECX := Native Model ID Enumeration Leaf;
 EDX := Native Model ID Enumeration Leaf;
 BREAK;
 EAX = 1BH:
 EAX := PCONFIG Information Enumeration Leaf; (* See "INPUT EAX = 1BH: Returns PCONFIG Information" on page 3-252. *)
 EBX := PCONFIG Information Enumeration Leaf;
 ECX := PCONFIG Information Enumeration Leaf;
 EDX := PCONFIG Information Enumeration Leaf;
 BREAK;
 EAX = 1CH:
 EAX := Last Branch Record Information Enumeration Leaf; (* See Table 3-8. *)
 EBX := Last Branch Record Information Enumeration Leaf;
 ECX := Last Branch Record Information Enumeration Leaf;

```

    EDX := Last Branch Record Information Enumeration Leaf;
BREAK;
EAX = 1DH:
    EAX := Tile Information Enumeration Leaf; (* See Table 3-8. *)
    EBX := Tile Information Enumeration Leaf;
    ECX := Tile Information Enumeration Leaf;
    EDX := Tile Information Enumeration Leaf;
BREAK;
EAX = 1EH:
    EAX := TMUL Information Enumeration Leaf; (* See Table 3-8. *)
    EBX := TMUL Information Enumeration Leaf;
    ECX := TMUL Information Enumeration Leaf;
    EDX := TMUL Information Enumeration Leaf;
BREAK;
EAX = 1FH:
    EAX := V2 Extended Topology Enumeration Leaf; (* See Table 3-8. *)
    EBX := V2 Extended Topology Enumeration Leaf;
    ECX := V2 Extended Topology Enumeration Leaf;
    EDX := V2 Extended Topology Enumeration Leaf;
BREAK;
EAX = 20H:
    EAX := Processor History Reset Sub-leaf; (* See Table 3-8. *)
    EBX := Processor History Reset Sub-leaf;
    ECX := Processor History Reset Sub-leaf;
    EDX := Processor History Reset Sub-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 3-8.*);
    EDX := Extended Feature Bits (* See Table 3-8. *);
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	<p>If the LOCK prefix is used.</p> <p>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.</p>
-----	---

CRC32—Accumulate CRC32 Value

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 38 F0 /r CRC32 r32, r/m8	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 REX 0F 38 F0 /r CRC32 r32, r/m8 ¹	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 0F 38 F1 /r CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 0F 38 F1 /r CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W 0F 38 F0 /r CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W 0F 38 F1 /r CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on r/m64.

NOTES:

1. 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)	N/A	N/A

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) 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 )
unsigned __int64 _mm_crc32_u64( unsigned __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.

CVTDDQ2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDDQ2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	AVX512VL AVX512F	Convert 2 packed signed doubleword integers from xmm2/m64/m32bcst to eight packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double precision floating-point values in the destination operand (the first operand).

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

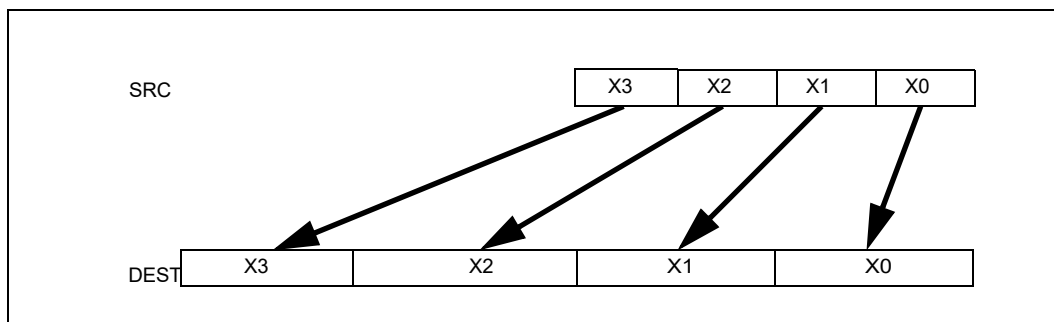


Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

Operation

VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

 i := j * 64

 k := j * 32

 IF k1[j] OR *no writemask*

 THEN DEST[i+63:i] :=

 Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[i+63:i] remains unchanged*

 ELSE ; zeroing-masking

 DEST[i+63:i] := 0

 FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] :=
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
    ENDIF;
  DEST[MAXVL-1:VL] := 0

```

VCVTDQ2PD (VEX.256 Encoded Version)

```

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])
DEST[191:128] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] := 0

```

VCVTDQ2PD (VEX.128 Encoded Version)

```

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])
DEST[MAXVL-1:128] := 0

```

CVTDQ2PD (128-bit Legacy SSE Version)

```

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])
DEST[MAXVL-1:128] (unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTDQ2PD __m512d __mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d __mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d __mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_cvtepi32_pd( __m128i src);
VCVTDQ2PD __m256d __mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d __mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d __mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m128d __mm_cvtepi32_pd( __m128i src)

```

Other Exceptions

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-51, “Type E5 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTDDQ2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single precision floating-point values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Operation**VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC); ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC); ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1

FI;

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] :=

Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] :=

Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])

ELSE

DEST[i+31:i] :=

Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 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[127:96])
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])
DEST[MAXVL-1:256] := 0

```

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[MAXVL-1:128] := 0

```

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[MAXVL-1:128] (unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTDQ2PS __m512 __mm512_cvtepi32_ps( __m512i a);
VCVTDQ2PS __m512 __mm512_mask_cvtepi32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTDQ2PS __m512 __mm512_maskz_cvtepi32_ps( __mmask16 k, __m512i a);
VCVTDQ2PS __m512 __mm512_cvt_roundepi32_ps( __m512i a, int r);
VCVTDQ2PS __m512 __mm512_mask_cvt_roundepi32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m512 __mm512_maskz_cvt_roundepi32_ps( __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m256 __mm256_mask_cvtepi32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTDQ2PS __m256 __mm256_maskz_cvtepi32_ps( __mmask8 k, __m256i a);
VCVTDQ2PS __m128 __mm_mask_cvtepi32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTDQ2PS __m128 __mm_maskz_cvtepi32_ps( __mmask8 k, __m128i a);
CVTDQ2PS __m256 __mm256_cvtepi32_ps( __m256i src)
CVTDQ2PS __m128 __mm_cvtepi32_ps( __m128i src)

```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed signed doubleword integers 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 or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^w-1 , where w represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

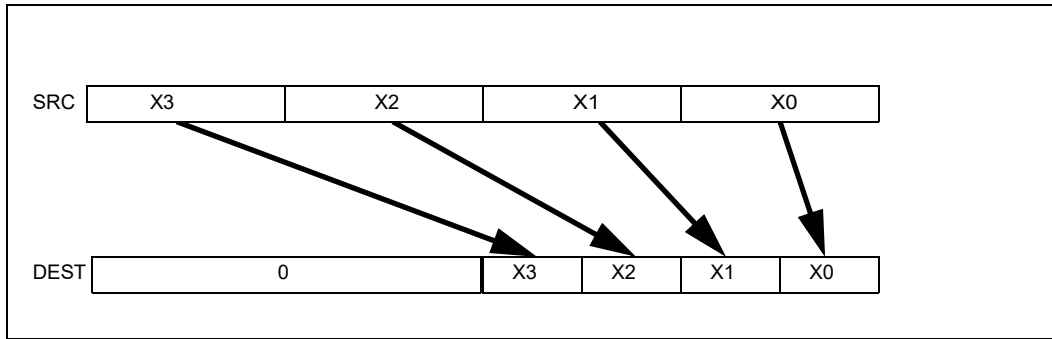


Figure 3-12. VCVTPD2DQ (VEX.256 encoded version)

Operation

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

k := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] :=

Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 0

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 32

k := j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] :=

Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])

ELSE

DEST[i+31:i] :=

Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 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[MAXVL-1:128] := 0

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[MAXVL-1:64] := 0

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[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2DQ __m256i _mm512_cvtpd_epi32( __m512d a);
VCVTPD2DQ __m256i _mm512_mask_cvtpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32( __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_cvt_roundpd_epi32( __m512d a, int r);
VCVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i _mm512_maskz_cvt_roundpd_epi32( __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i _mm256_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32( __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm_maskz_cvtpd_epi32( __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm256_cvtpd_epi32( __m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32( __m128d src)

```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2D /r CVTPD2PI mm, xmm/m128	RM	V/V	SSE2	Convert two packed double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm.

Instruction Operand Encoding

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

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 23-4, “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 Floating-Point Values to Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four single precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two single precision floating-point values in xmm1 with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four single precision floating-point values in xmm1 with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight single precision floating-point values in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight 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 or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

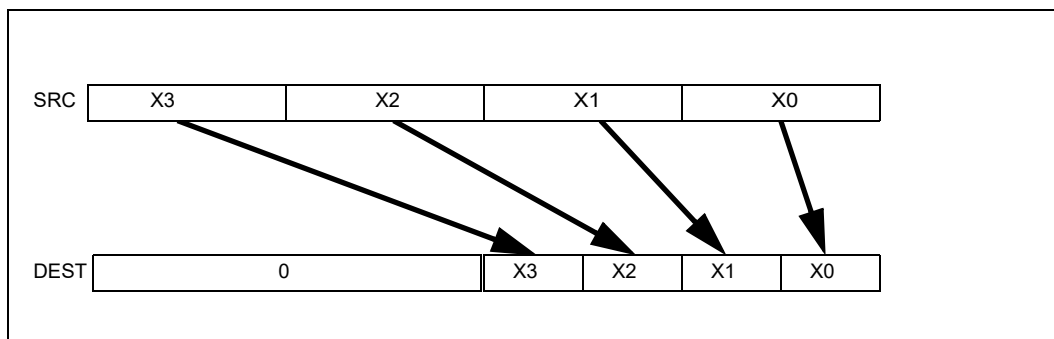


Figure 3-13. VCVTPD2PS (VEX.256 encoded version)

Operation

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

k := j * 64

IF k1[j] OR *no writemask*

THEN

DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 0

VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI;
    ELSE
      FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

```

VCVTPD2PS (VEX.256 Encoded Version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
DEST[MAXVL-1:128] := 0

```

VCVTPD2PS (VEX.128 Encoded Version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] := 0

```

CVTPD2PS (128-bit Legacy SSE Version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2PS __m256 __mm512_cvtpd_ps( __m512d a);
VCVTPD2PS __m256 __mm512_mask_cvtpd_ps( __m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_maskz_cvtpd_ps( __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_cvt_roundpd_ps( __m512d a, int r);
VCVTPD2PS __m256 __mm512_mask_cvt_roundpd_ps( __m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 __mm512_maskz_cvt_roundpd_ps( __mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 __mm256_mask_cvtpd_ps( __m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm256_maskz_cvtpd_ps( __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm_mask_cvtpd_ps( __m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm_maskz_cvtpd_ps( __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm256_cvtpd_ps( __m256d a);
CVTPD2PS __m128 __mm_cvtpd_ps( __m128d a);

```

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPI2PD—Convert Packed Dword Integers to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /r CVTPI2PD xmm, mm/m64 ¹	RM	Valid	Valid	Convert two packed signed doubleword integers from mm/mem64 to two packed double precision floating-point values in xmm.

NOTES:

1. 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)	N/A	N/A

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 23-6, “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 Floating-Point Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 2A /r CVTPI2PS xmm, mm/m64	RM	Valid	Valid	Convert two signed doubleword integers from mm/m64 to two single precision floating-point values in xmm.

Instruction Operand Encoding

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

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 23-5, “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.

CVTTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen 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 or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^w-1 , where w represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation**VCVTPS2DQ (Encoded Versions) When SRC Operand is a Register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] :=

Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VCVTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO 15

i := j * 32

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] :=

Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])

ELSE

DEST[i+31:i] :=

Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VCVTPS2DQ (VEX.256 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[159:128] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])

```

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[MAXVL-1:128] := 0

```

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[MAXVL-1:128] (unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPS2DQ __m512i __mm512_cvtps_epi32( __m512 a);
VCVTPS2DQ __m512i __mm512_mask_cvtps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i __mm512_maskz_cvtps_epi32( __mmask16 k, __m512 a);
VCVTPS2DQ __m512i __mm512_cvt_roundps_epi32( __m512 a, int r);
VCVTPS2DQ __m512i __mm512_mask_cvt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m512i __mm512_maskz_cvt_roundps_epi32( __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m256i __mm256_mask_cvtps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2DQ __m256i __mm256_maskz_cvtps_epi32( __mmask8 k, __m256 a);
VCVTPS2DQ __m128i __mm_mask_cvtps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i __mm_maskz_cvtps_epi32( __mmask8 k, __m128 a);
VCVTPS2DQ __m256i __mm256_cvtps_epi32( __m256 a);
CVTPS2DQ __m128i __mm_cvtps_epi32( __m128 a);

```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 5A /r CVTTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.128.OF.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.
VEX.256.OF.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values in xmm2/m128 to four packed double precision floating-point values in ymm1.
EVEX.128.OF.WO 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	AVX512VL AVX512F	Convert two packed single precision floating-point values in xmm2/m64/m32bcst to packed double precision floating-point values in xmm1 with writemask k1.
EVEX.256.OF.WO 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL	Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed double precision floating-point values in ymm1 with writemask k1.
EVEX.512.OF.WO 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	B	V/V	AVX512F	Convert eight packed single precision floating-point values in ymm2/m256/b32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Half	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed single precision floating-point values in the source operand (second operand) to two, four or eight packed double precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

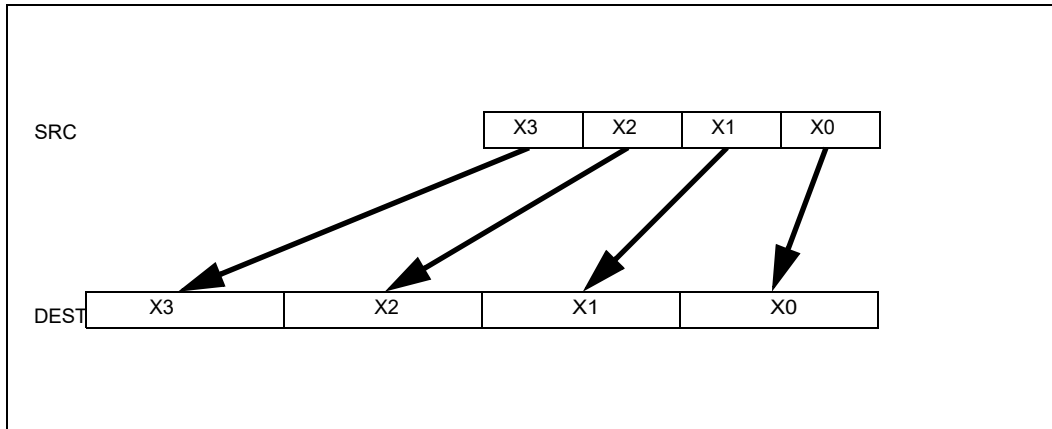


Figure 3-14. CVTSP2PD (VEX.256 encoded version)

Operation

VCVTSP2PD (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

 i := j * 64

 k := j * 32

 IF k1[j] OR *no writemask*

 THEN DEST[i+63:i] :=

 Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[i+63:i] remains unchanged*

 ELSE ; zeroing-masking

 DEST[i+63:i] := 0

 FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VCVTSP2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

 i := j * 64

 k := j * 32

 IF k1[j] OR *no writemask*

 THEN

 IF (EVEX.b = 1)

 THEN

 DEST[i+63:i] :=

 Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])

 ELSE

 DEST[i+63:i] :=

 Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])

 FI;

 ELSE

```

        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

VCVTPS2PD (VEX.256 Encoded Version)

```

DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] := 0

```

VCVTPS2PD (VEX.128 Encoded Version)

```

DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0

```

CVTPS2PD (128-bit Legacy SSE Version)

```

DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPS2PD __m512d __mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d __mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d __mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTPS2PD __m512d __mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTPS2PD __m512d __mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d __mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d __mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d __mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTPS2PD __m128d __mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d __mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTPS2PD __m256d __mm256_cvtps_pd( __m128 a)
CVTPS2PD __m128d __mm_cvtps_pd( __m128 a)

```

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 2D /r CVTPS2PI mm, xmm/m64	RM	Valid	Valid	Convert two packed single precision floating-point values from xmm/m64 to two packed signed doubleword integers in mm.

Instruction Operand Encoding

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

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 23-5, “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 Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r ¹ VCVTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r ¹ VCVTSD2SI r64, xmm1/m64	A	V/N.E. ²	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LLIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	B	V/V	AVX512F	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LLIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	B	V/N.E. ²	AVX512F	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the 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 exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SI (EVEX Encoded Version)

```

IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE    DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI

```

(V)CVTSD2SI

```

IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE
        DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI;

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTSD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double precision floating-point value in xmm2/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1, xmm2, xmm3/m64	B	V/V	AVX	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2.
EVEX.LLIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a double precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single precision floating-point value in the destination operand.

When the “convert-from” 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. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SS (EVEX Encoded Version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

VCVTSD2SS (VEX.128 Encoded Version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

CVTSD2SS (128-bit Legacy SSE Version)

```

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSD2SS __m128_mm_mask_cvtssd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128_mm_maskz_cvtssd_ss(__mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128_mm_cvt_roundssd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128_mm_mask_cvt_roundssd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128_mm_maskz_cvt_roundssd_ss(__mmask8 k, __m128 a, __m128d b, int r);
CVTSD2SS __m128_mm_cvtssd_ss(__m128 a, __m128d b)

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	A	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	A	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	B	V/N.E. ¹	AVX	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	C	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.
EVEX.LLIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	C	V/N.E. ¹	AVX512F	Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” 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.

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: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD (EVEX Encoded Version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
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]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

VCVTSI2SD (VEX.128 Encoded Version)

```

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]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

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[MAXVL-1:64] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d _mm_cvtsi32_sd(__m128d a, int b)

```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” if W1; else see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions,” if W1; else see Table 2-59, “Type E10NF Class Exception Conditions.”

CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	A	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	A	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
VEX.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	B	V/N.E. ¹	AVX	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	C	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.
EVEX.LLIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	C	V/N.E. ¹	AVX512F	Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a single precision floating-point value in the destination operand (first operand). The “convert-from” 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 or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (EVEX Encoded Version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
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[MAXVL-1:128] := 0

```

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[MAXVL-1:128] := 0

```

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]);
FI;
DEST[MAXVL-1:32] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSI2SS __m128 _mm_cvtsi32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);

```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single precision floating-point value in xmm2/m32 to one double precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2.
EVEX.LLIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Converts a single precision floating-point value in the “convert-from” source operand to a double precision floating-point value in the destination operand. When the “convert-from” 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.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SD (EVEX Encoded Version)

IF k1[0] or *no writemask*

THEN DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[63:0] = 0

FI;

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0

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[MAXVL-1:128] := 0

CVTSS2SD (128-bit Legacy SSE Version)

DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);

DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);

VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a, __m128 b, int r);

VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);

VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a, __m128 b);

VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a, __m128 b);

CVTSS2SD __m128d _mm_cvtss_sd(__m128d a, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r ¹ VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r ¹ VCVTSS2SI r64, xmm1/m32	A	V/N.E. ²	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LLIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	B	V/V	AVX512F	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LLIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	B	V/N.E. ²	AVX512F	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

- Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
- VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the 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 or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^w-1 , where w represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SI (EVEX Encoded Version)

```

IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-bit Mode and OperandSize = 64
    THEN
        DEST[63:0] := 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;

```

(V)CVTSS2SI (Legacy and VEX.128 Encoded Version)

```

IF 64-bit Mode and OperandSize = 64
    THEN
        DEST[63:0] := 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

```

VCVTSS2SI int _mm_cvtss_i32( __m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32( __m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64( __m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64( __m128 a, int r);

```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	B	V/V	AVX512F	Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight 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.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

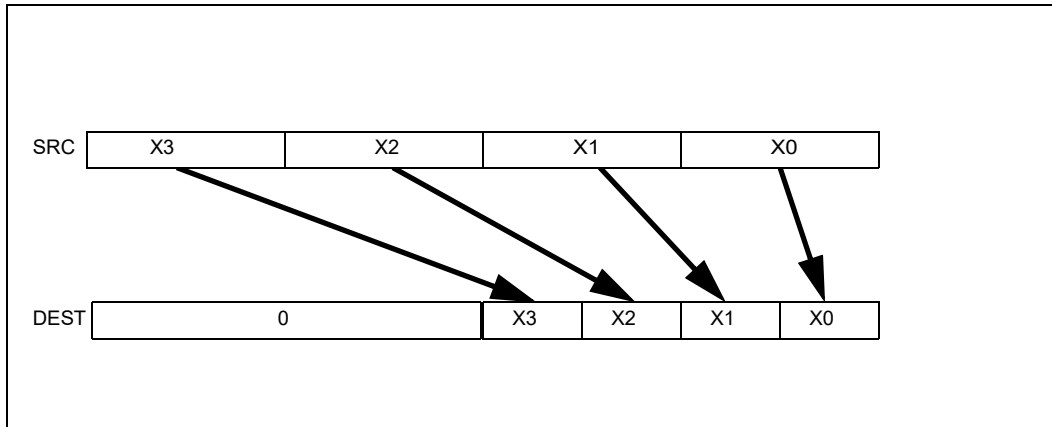


Figure 3-15. VCVTTPD2DQ (VEX.256 encoded version)

Operation

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
      Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                           ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

```

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI;
    ENDIF;
  ENDFOR
DEST[MAXVL-1:VL/2] := 0

```

VCVTTPD2DQ (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[MAXVL-1:128] := 0

```

VCVTTPD2DQ (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[MAXVL-1:64] := 0

```

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[MAXVL-1:128] (unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPD2DQ __m256i __mm512_cvttpd_epi32( __m512d a);
VCVTPD2DQ __m256i __mm512_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_cvtt_roundpd_epi32( __m512d a, int sae);
VCVTPD2DQ __m256i __mm512_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTPD2DQ __m256i __mm512_maskz_cvtt_roundpd_epi32( __mmask8 k, __m512d a, int sae);
VCVTPD2DQ __m128i __mm256_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm_maskz_cvttpd_epi32( __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm256_cvttpd_epi32( __m256d src);
CVTPD2DQ __m128i __mm_cvttpd_epi32( __m128d src);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTTPD2PI—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTTPD2PI mm, xmm/m128	RM	Valid	Valid	Convert two packed double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm using truncation.

Instruction Operand Encoding

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

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 CVTTPD2PI 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

CVTTPD1PI __m64 _mm_cvttpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Table 23-4, "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.

CVTTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	B	V/V	AVX512F	Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen 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.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation**VCVTTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
      Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

VCVTTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j := 0 TO 15
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

VCVTTTPS2DQ (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])

```

VCVTTPS2DQ (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[MAXVL-1:128] := 0

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[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2DQ __m512i __mm512_cvttps_epi32(__m512 a);
 VCVTTPS2DQ __m512i __mm512_mask_cvttps_epi32(__m512i s, __mmask16 k, __m512 a);
 VCVTTPS2DQ __m512i __mm512_maskz_cvttps_epi32(__mmask16 k, __m512 a);
 VCVTTPS2DQ __m512i __mm512_cvtt_roundps_epi32(__m512 a, int sae);
 VCVTTPS2DQ __m512i __mm512_mask_cvtt_roundps_epi32(__m512i s, __mmask16 k, __m512 a, int sae);
 VCVTTPS2DQ __m512i __mm512_maskz_cvtt_roundps_epi32(__mmask16 k, __m512 a, int sae);
 VCVTTPS2DQ __m256i __mm256_mask_cvttps_epi32(__m256i s, __mmask8 k, __m256 a);
 VCVTTPS2DQ __m256i __mm256_maskz_cvttps_epi32(__mmask8 k, __m256 a);
 VCVTTPS2DQ __m128i __mm_mask_cvttps_epi32(__m128i s, __mmask8 k, __m128 a);
 VCVTTPS2DQ __m128i __mm_maskz_cvttps_epi32(__mmask8 k, __m128 a);
 VCVTTPS2DQ __m256i __mm256_cvttps_epi32(__m256 a);
 CVTTPS2DQ __m128i __mm_cvttps_epi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTTPS2PI—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2C /r CVTTPS2PI mm, xmm/m64	RM	Valid	Valid	Convert two single precision floating-point values from xmm/m64 to two signed doubleword signed integers in mm using truncation.

Instruction Operand Encoding

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

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 23-5, “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 Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r ¹ VCVTTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r ¹ VCVTTSD2SI r64, xmm1/m64	B	V/N.E. ²	AVX	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	B	V/V	AVX512F	Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	B	V/N.E. ²	AVX512F	Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the 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 exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) 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.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
 Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSD2SI (All Versions)

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);

ELSE

DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);

FI;

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int __mm_cvttssd_i32(__m128d a);

VCVTTSD2SI int __mm_cvtt_roundssd_i32(__m128d a, int sae);

VCVTTSD2SI __int64 __mm_cvttssd_i64(__m128d a);

VCVTTSD2SI __int64 __mm_cvtt_roundssd_i64(__m128d a, int sae);

CVTTSD2SI int __mm_cvttssd_si32(__m128d a);

CVTTSD2SI __int64 __mm_cvttssd_si64(__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

CVTTSS2SI—Convert With Truncation Scalar Single Precision Floating-Point Value to Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r ¹ VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r ¹ VCVTSS2SI r64, xmm1/m32	A	V/N.E. ²	AVX	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LLIG.F3.0F.W0 2C /r VCVTSS2SI r32, xmm1/m32{sae}	B	V/V	AVX512F	Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LLIG.F3.0F.W1 2C /r VCVTSS2SI r64, xmm1/m32{sae}	B	V/N.E. ²	AVX512F	Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

- Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
- For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the 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 or 80000000_00000000H if operand size is 64 bits) 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.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSS2SI (All Versions)

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

VCVTTSS2SI int __mm_cvtss_i32(__m128 a);

VCVTTSS2SI int __mm_cvt_roundss_i32(__m128 a, int sae);

VCVTTSS2SI __int64 __mm_cvtss_i64(__m128 a);

VCVTTSS2SI __int64 __mm_cvt_roundss_i64(__m128 a, int sae);

CVTTSS2SI int __mm_cvtss_si32(__m128 a);

CVTTSS2SI __int64 __mm_cvtss_si64(__m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
99	CWD	Z0	Valid	Valid	DX:AX := sign-extend of AX.
99	CDQ	Z0	Valid	Valid	EDX:EAX := sign-extend of EAX.
REX.W + 99	CQO	Z0	Valid	N.E.	RDX:RAX:= sign-extend of RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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); FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
        RDX := SignExtend(RAX); FI;
FI;
```

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	Z0	Invalid	Valid	Decimal adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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	Z0	Invalid	Valid	Decimal adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 <i>r/m8</i>	M	Valid	Valid	Decrement <i>r/m8</i> by 1.
REX + FE /1	DEC <i>r/m8</i>	M	Valid	N.E.	Decrement <i>r/m8</i> by 1.
FF /1	DEC <i>r/m16</i>	M	Valid	Valid	Decrement <i>r/m16</i> by 1.
FF /1	DEC <i>r/m32</i>	M	Valid	Valid	Decrement <i>r/m32</i> by 1.
REX.W + FF /1	DEC <i>r/m64</i>	M	Valid	N.E.	Decrement <i>r/m64</i> by 1.
48+rw	DEC <i>r16</i>	O	N.E.	Valid	Decrement <i>r16</i> by 1.
48+rd	DEC <i>r32</i>	O	N.E.	Valid	Decrement <i>r32</i> 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 (<i>r</i> , <i>w</i>)	N/A	N/A	N/A
O	opcode + <i>rd</i> (<i>r</i> , <i>w</i>)	N/A	N/A	N/A

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 ¹	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:

1. 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)	N/A	N/A	N/A

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 3-15.

Table 3-15. 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; FI; (* 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;
        FI;
    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;
            FI;
        FI;
    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;
            FI;
        FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp := RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
                ELSE
                    RAX := temp;
                    RDX := RDX:RAX MOD 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 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 Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double precision floating-point values in xmm1 by packed double precision floating-point values in xmm2/mem.
VEX.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/mem.
VEX.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/mem.
EVEX.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Divide packed double precision floating-point values in zmm2 by packed double precision floating-point values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD divide of the double precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand (the second 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. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

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

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.

Operation**VDIVPD (EVEX Encoded Versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC); ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures
Software Developer's Manual, Volume 1

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

DEST[i+63:i] := SRC1[i+63:i] / SRC2[63:0]

ELSE

DEST[i+63:i] := SRC1[i+63:i] / SRC2[i+63:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VDIVPD (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]

DEST[MAXVL-1:256] := 0;

VDIVPD (VEX.128 Encoded Version)

DEST[63:0] := SRC1[63:0] / SRC2[63:0]

DEST[127:64] := SRC1[127:64] / SRC2[127:64]

DEST[MAXVL-1:128] := 0;

DIVPD (128-bit Legacy SSE Version)

DEST[63:0] := SRC1[63:0] / SRC2[63:0]

DEST[127:64] := SRC1[127:64] / SRC2[127:64]

DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

```

VDIVPD __m512d _mm512_div_pd( __m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd( __mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d _mm512_div_round_pd( __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_maskz_div_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d _mm256_div_pd( __m256d a, __m256d b);
DIVPD __m128d _mm_div_pd( __m128d a, __m128d b);

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

DIVPS—Divide Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single precision floating-point values in xmm1 by packed single precision floating-point values in xmm2/mem.
VEX.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/mem.
VEX.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/mem.
EVEX.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F	Divide packed single precision floating-point values in zmm2 by packed single precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a SIMD divide of the four, eight or sixteen packed single precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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 (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VDIVPS (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

DEST[i+31:i] := SRC1[i+31:i] / SRC2[31:0]

ELSE

DEST[i+31:i] := SRC1[i+31:i] / SRC2[i+31:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VDIVPS (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].

DEST[MAXVL-1:256] := 0;

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[MAXVL-1:128] := 0

DIVPS (128-bit Legacy SSE Version)

$\text{DEST}[31:0] := \text{SRC1}[31:0] / \text{SRC2}[31:0]$
 $\text{DEST}[63:32] := \text{SRC1}[63:32] / \text{SRC2}[63:32]$
 $\text{DEST}[95:64] := \text{SRC1}[95:64] / \text{SRC2}[95:64]$
 $\text{DEST}[127:96] := \text{SRC1}[127:96] / \text{SRC2}[127:96]$
 $\text{DEST}[\text{MAXVL}-1:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`VDIVPS __m512 __mm512_div_ps(__m512 a, __m512 b);`
`VDIVPS __m512 __mm512_mask_div_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);`
`VDIVPS __m512 __mm512_maskz_div_ps(__mmask16 k, __m512 a, __m512 b);`
`VDIVPD __m256d __mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);`
`VDIVPD __m256d __mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);`
`VDIVPD __m128d __mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);`
`VDIVPD __m128d __mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);`
`VDIVPS __m512 __mm512_div_round_ps(__m512 a, __m512 b, int);`
`VDIVPS __m512 __mm512_mask_div_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);`
`VDIVPS __m512 __mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int);`
`VDIVPS __m256 __mm256_div_ps(__m256 a, __m256 b);`
`DIVPS __m128 __mm_div_ps(__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

DIVSD—Divide Scalar Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double precision floating-point value in xmm1 by low double precision floating-point value in xmm2/m64.
VEX.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.
EVEX.LLIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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 are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSD (EVEX Encoded Version)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

VDIVSD (VEX.128 Encoded Version)

```

DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

DIVSD (128-bit Legacy SSE Version)

```

DEST[63:0] := DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd(__m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd(__m128d a, __m128d b);

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

DIVSS—Divide Scalar Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single precision floating-point value in xmm1 by low single precision floating-point value in xmm2/m32.
VEX.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.
EVEX.LLIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

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.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSS (EVEX Encoded Version)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] := SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[31:0] := 0
        FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

VDIVSS (VEX.128 Encoded Version)

```

DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

DIVSS (128-bit Legacy SSE Version)

```

DEST[31:0] := DEST[31:0] / SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss(__mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss(__m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss(__mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);

```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

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 xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Selectively multiply packed double precision floating-point values from xmm1 with packed double precision floating-point values from xmm2, add and selectively store the packed double precision floating-point values to xmm1.
VEX.128.66.0F3A.WIG 41 /r ib VDPPD xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Selectively multiply packed double precision floating-point values from xmm2 with packed double precision floating-point values from xmm3, add and selectively store the packed double precision floating-point values to xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
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 (MAXVL-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 (MAXVL-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; FI;
IF (imm8[5] = 1)
    THEN Temp1[127:64] := DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] := +0.0; FI;
/* 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; FI;
IF (imm8[1] = 1)
    THEN DEST[127:64] := Temp2[63:0];
    ELSE DEST[127:64] := +0.0; FI;

```

DPPD (128-bit Legacy SSE Version)

```

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)

```

VDPPD (VEX.128 Encoded Version)

```

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-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 Table 2-19, “Type 2 Class Exception Conditions,” 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 xmm1, xmm2/m128, imm8	RMI	V/V	SSE4_1	Selectively multiply packed single precision floating-point values from xmm1 with packed single precision floating-point values from xmm2, add and selectively store the packed single precision floating-point values or zero values to xmm1.
VEX.128.66.0F3A.WIG 40 /r ib VDPPS xmm1,xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Multiply packed single precision floating-point values from xmm1 with packed single precision floating-point values from xmm2/mem selectively add and store to xmm1.
VEX.256.66.0F3A.WIG 40 /r ib VDPPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Multiply packed single precision floating-point values from ymm2 with packed single precision floating-point values from ymm3/mem, selectively add pairs of elements and store to ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
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 (MAXVL-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 (MAXVL-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; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] := DEST[63:32] * SRC[63:32]; // update SIMD exception flags
    ELSE Temp1[63:32] := +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] := DEST[95:64] * SRC[95:64]; // update SIMD exception flags
    ELSE Temp1[95:64] := +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] := DEST[127:96] * SRC[127:96]; // update SIMD exception flags
    ELSE Temp1[127:96] := +0.0; FI;

```

```

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; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] := Temp4[31:0];
    ELSE DEST[63:32] := +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] := Temp4[31:0];
    ELSE DEST[95:64] := +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] := Temp4[31:0];
    ELSE DEST[127:96] := +0.0; FI;

```

DPPS (128-bit Legacy SSE Version)

```

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)

```

VDPPS (VEX.128 Encoded Version)

```

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-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 Table 2-19, “Type 2 Class Exception Conditions.”

EMMS—Empty MMX Technology State

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF 77	EMMS	Z0	Valid	Valid	Set the x87 FPU tag word to empty.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 FPU 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.

ENCODEKEY128—Encode 128-Bit Key With Key Locker

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 FA 11:rrr:bbb ENCODEKEY128 r32, r32, <XMM0-2>, <XMM4-6>	A	V/V	AESKLE	Wrap a 128-bit AES key from XMM0 into a key handle and output handle in XMM0—2.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operands 4—5	Operands 6—7
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	Implicit XMM0 (r, w)	Implicit XMM1—2 (w)	Implicit XMM4—6 (w)

Description

The ENCODEKEY128¹ instruction wraps a 128-bit AES key from the implicit operand XMM0 into a key handle that is then stored in the implicit destination operands XMM0–2.

The explicit source operand specifies handle restrictions, if any.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

Operation

ENCODEKEY128

#GP (0) if a reserved bit² in SRC[31:0] is set

InputKey[127:0] := XMM0;

KeyMetadata[2:0] = SRC[2:0];

KeyMetadata[23:3] = 0; // Reserved for future usage

KeyMetadata[27:24] = 0; // KeyType is AES-128 (value of 0)

KeyMetadata[127:28] = 0; // Reserved for future usage

// KeyMetadata is the AAD input and InputKey is the Plaintext input for WrapKey128

Handle[383:0] := WrapKey128(InputKey[127:0], KeyMetadata[127:0], lWKey.Integrity Key[127:0], lWKey.Encryption Key[255:0]);

DEST[0] := lWKey.NoBackup;

DEST[4:1] := lWKey.KeySource[3:0];

DEST[31:5] = 0;

XMM0 := Handle[127:0]; // AAD

XMM1 := Handle[255:128]; // Integrity Tag

XMM2 := Handle[383:256]; // CipherText

XMM4 := 0; // Reserved for future usage

XMM5 := 0; // Reserved for future usage

XMM6 := 0; // Reserved for future usage

RFLAGS.OF, SF, ZF, AF, PF, CF := 0;

Flags Affected

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

2. SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.

Intel C/C++ Compiler Intrinsic Equivalent

ENCODKEY128 unsigned int _mm_encodekey128_u32(unsigned int htype, __m128i key, void* h);

Exceptions (All Operating Modes)

- #GP If reserved bit is set in source register value.
- #UD If the LOCK prefix is used.
If CPUID.07H:ECX.KL [bit 23] = 0.
If CR4.KL = 0.
If CPUID.19H:EBX.AESKLE [bit 0] = 0.
If CR0.EM = 1.
If CR4.OSFXSR = 0.
- #NM If CR0.TS = 1.

ENCODEKEY256—Encode 256-Bit Key With Key Locker

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 FB 11:rrr:bbb ENCODEKEY256 r32, r32 <XMM0-6>	A	V/V	AESKLE	Wrap a 256-bit AES key from XMM1:XMM0 into a key handle and store it in XMM0—3.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operands 3—4	Operands 5—9
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	Implicit XMM0—1 (r, w)	Implicit XMM2—6 (w)

Description

The ENCODEKEY256¹ instruction wraps a 256-bit AES key from the implicit operand XMM1:XMM0 into a key handle that is then stored in the implicit destination operands XMM0-3.

The explicit source operand is a general-purpose register and specifies what handle restrictions should be built into the handle.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

Operation

ENCODEKEY256

#GP (0) if a reserved bit² in SRC[31:0] is set

InputKey[255:0] := XMM1:XMM0;

KeyMetadata[2:0] = SRC[2:0];

KeyMetadata[23:3] = 0; // Reserved for future usage

KeyMetadata[27:24] = 1; // KeyType is AES-256 (value of 1)

KeyMetadata[127:28] = 0; // Reserved for future usage

// KeyMetadata is the AAD input and InputKey is the Plaintext input for WrapKey256

Handle[511:0] := WrapKey256(InputKey[255:0], KeyMetadata[127:0], lWKey.Integrity Key[127:0], lWKey.Encryption Key[255:0]);

DEST[0] := lWKey.NoBackup;

DEST[4:1] := lWKey.KeySource[3:0];

DEST[31:5] = 0;

XMM0 := Handle[127:0]; // AAD

XMM1 := Handle[255:128]; // Integrity Tag

XMM2 := Handle[383:256]; // CipherText[127:0]

XMM3 := Handle[511:384]; // CipherText[255:128]

XMM4 := 0; // Reserved for future usage

XMM5 := 0; // Reserved for future usage

XMM6 := 0; // Reserved for future usage

RFLAGS.OF, SF, ZF, AF, PF, CF := 0;

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

2. SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.

Flags Affected

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

Intel C/C++ Compiler Intrinsic Equivalent

```
ENCODKEY256 unsigned int __mm_encodekey256_u32(unsigned int htype, __m128i key_lo, __m128i key_hi, void* h);
```

Exceptions (All Operating Modes)

#GP	If reserved bit is set in source register value.
#UD	If the LOCK prefix is used. If CPUID.07H:ECX.KL [bit 23] = 0. If CR4.KL = 0. If CPUID.19H:EBX.AESKLE [bit 0] = 0. If CR0.EM = 1. If CR4.OSFXSR = 0.
#NM	If CR0.TS = 1.

ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FB ENDBR32	Z0	V/V	CET_IBT	Terminate indirect branch in 32-bit and compatibility mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A	N/A

Description

Terminate an indirect branch in 32 bit and compatibility mode.

Operation

IF EndbranchEnabled(CPL) & (IA32_EFER.LMA = 0 | (IA32_EFER.LMA=1 & CS.L = 0))

IF CPL = 3

THEN

IA32_U_CET.TRACKER = IDLE

IA32_U_CET.SUPPRESS = 0

ELSE

IA32_S_CET.TRACKER = IDLE

IA32_S_CET.SUPPRESS = 0

FI;

FI;

Flags Affected

None.

Exceptions

None.

ENDBR64—Terminate an Indirect Branch in 64-bit Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FA ENDBR64	ZO	V/V	CET_IBT	Terminate indirect branch in 64-bit mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

Terminate an indirect branch in 64 bit mode.

Operation

IF EndbranchEnabled(CPL) & IA32_EFER.LMA = 1 & CS.L = 1
 IF CPL = 3
 THEN
 IA32_U_CET.TRACKER = IDLE
 IA32_U_CET.SUPPRESS = 0
 ELSE
 IA32_S_CET.TRACKER = IDLE
 IA32_S_CET.SUPPRESS = 0
 FI;
FI;

Flags Affected

None.

Exceptions

None.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C8 iw 00	ENTER imm16, 0	II	Valid	Valid	Create a stack frame for a procedure.
C8 iw 01	ENTER imm16, 1	II	Valid	Valid	Create a stack frame with a nested pointer for a procedure.
C8 iw ib	ENTER imm16, imm8	II	Valid	Valid	Create a stack frame with nested pointers for a procedure.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	iw	imm8	N/A	N/A

Description

Creates a stack frame (comprising of space for dynamic storage and 1-32 frame pointer storage) for a procedure. The first operand (imm16) specifies the size of the dynamic storage in the stack frame (that is, the number of bytes of dynamically allocated on the stack for the procedure). The second operand (imm8) gives the lexical nesting level (0 to 31) of the procedure. The nesting level (imm8 mod 32) and the OperandSize attribute determine the size in bytes of the storage space for frame pointers.

The nesting level determines the number of frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. The default size of the frame pointer is the StackAddrSize attribute, but can be overridden using the 66H prefix. Thus, the OperandSize attribute determines the size of each frame pointer that will be copied into the stack frame and the data being transferred from SP/ESP/RSP register into the BP/EBP/RBP register.

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. Use of 66H prefix changes frame pointer operand size to 16 bits.

When the 66H prefix is used and causing the OperandSize attribute to be less than the StackAddrSize, software is responsible for the following:

- The companion LEAVE instruction must also use the 66H prefix,
- The value in the RBP/EBP register prior to executing “66H ENTER” must be within the same 16KByte region of the current stack pointer (RSP/ESP), such that the value of RBP/EBP after “66H ENTER” remains a valid address in the stack. This ensures “66H LEAVE” can restore 16-bits of data from the stack.

Operation

```

AllocSize := imm16;
NestingLevel := imm8 MOD 32;
IF (OperandSize = 64)
    THEN
        Push(RBP); (* RSP decrements by 8 *)
        FrameTemp := RSP;
    ELSE IF OperandSize = 32
        THEN
            Push(EBP); (* (E)SP decrements by 4 *)
            FrameTemp := ESP; FI;
    ELSE (* OperandSize = 16 *)
        Push(BP); (* RSP or (E)SP decrements by 2 *)
        FrameTemp := SP;
FI;

IF NestingLevel = 0
    THEN GOTO CONTINUE;
FI;

IF (NestingLevel > 1)
    THEN FOR i := 1 to (NestingLevel - 1)
        DO
            IF (OperandSize = 64)
                THEN
                    RBP := RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            THEN
                                EBP := EBP - 4;
                                Push([EBP]); (* Doubleword push *)
                            ELSE (* StackSize = 16 *)
                                BP := BP - 4;
                                Push([BP]); (* Doubleword push *)
                            FI;
                        FI;
                    ELSE (* OperandSize = 16 *)
                        IF StackSize = 64
                            THEN
                                RBP := RBP - 2;
                                Push([RBP]); (* Word push *)
                            ELSE IF StackSize = 32
                                THEN
                                    EBP := EBP - 2;
                                    Push([EBP]); (* Word push *)
                                ELSE (* StackSize = 16 *)
                                    BP := BP - 2;
                                    Push([BP]); (* Word push *)
                                FI;
                            FI;
                        FI;
                    FI;
            FI;
        OD;
    FI;
    IF (OperandSize = 64) (* nestinglevel 1 *)

```

```

THEN
    Push(FrameTemp); (* Quadword push and RSP decrements by 8 *)
ELSE IF OperandSize = 32
    THEN
        Push(FrameTemp); FI; (* Doubleword push and (E)SP decrements by 4 *)
    ELSE (* OperandSize = 16 *)
        Push(FrameTemp); (* Word push and RSP|ESP|SP decrements by 2 *)
FI;
CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP := FrameTemp;
        RSP := RSP - AllocSize;
    ELSE IF OperandSize = 32
        THEN
            EBP := FrameTemp;
            ESP := ESP - AllocSize; FI;
    ELSE (* OperandSize = 16 *)
        BP := FrameTemp[15:1]; (* Bits 16 and above of applicable RBP/EBP are unmodified *)
        SP := SP - AllocSize;
FI;
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.

ENQCMD—Enqueue Command

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 38 F8 !{(11):rrr:bbb ENQCMD r32/r64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte user command from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The ENQCMD instruction allows software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).
Enqueue registers expect writes to have the following format:

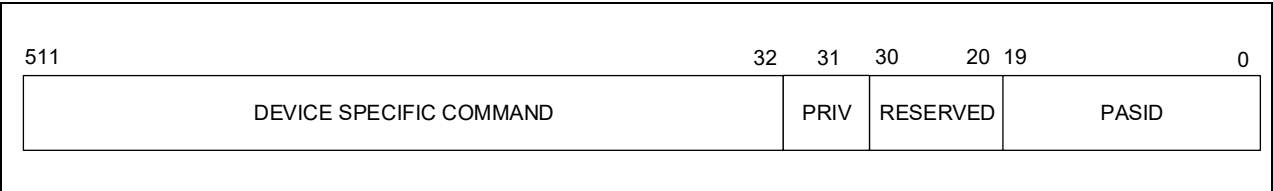


Figure 3-16. 64-Byte Data Written to Enqueue Registers

Bits 19:0 convey the process address space identifier (PASID), a value which system software may assign to individual software threads. Bit 31 contains privilege identification (0 = user; 1 = supervisor). Devices implementing enqueue registers may use these two values along with a device-specific command in the upper 60 bytes.

The ENQCMD instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically. Bits 31:0 of the source operand must be zero.

The instruction then formats those 64 bytes into **command data** with a format consistent with that given in Figure 3-16:

- Command[19:0] get IA32_PASID[19:0].¹
- Command[30:20] are zero.
- Command[31] is 0 (indicating user; this value is used regardless of CPL).
- Command[511:32] get bits 511:32 of the source operand that was read from memory.

The ENQCMD instruction uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).² The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

1. It is expected that system software will load the IA32_PASID MSR so that bits 19:0 contain the PASID of the current software thread. The MSR's valid bit, IA32_PASID[31], must be 1. For additional details on the IA32_PASID MSR, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

2. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMD instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons. This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

Availability of the ENQCMD instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.(EAX=07H, ECX=0H):ECX[bit 29]).

Operation

```
IF IA32_PASID[31] = 0
    THEN #GP;
ELSE
    COMMAND := (SRC & ~FFFFFFFFH) | (IA32_PASID & FFFFFFFFH);
    DEST := COMMAND;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ENQCMD int_enqcmd(void *dst, const void *src)
```

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

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 destination linear address is not aligned to a 64-byte boundary. If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR. If bits 31:0 of the source operand are not all zero.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0. If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP
 - If any part of the operand lies outside the effective address space from 0 to FFFFH.
 - If destination linear address is not aligned to a 64-byte boundary.
 - If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
 - If bits 31:0 of the source operand are not all zero.
- #UD
 - If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
 - If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real-address mode. Additionally:

- #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 non-canonical form.
- #GP(0)
 - If the memory address is in non-canonical form.
 - If destination linear address is not aligned to a 64-byte boundary.
 - If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
 - If bits 31:0 of the source operand are not all zero.
- #PF(fault-code) For a page fault.
- #UD
 - If CPUID.07H.0H:ECX.ENQCMD[bit 29].
 - If the LOCK prefix is used.

ENQCMDS—Enqueue Command Supervisor

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F8 !{11};rrr:bbb ENQCMDS r32/r64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte command with PASID from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

The ENQCMDS instruction allows system software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the format given in Figure 3-16 and explained in the section on “ENQCMD—Enqueue Command.”

The ENQCMDS instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically. Bits 30:20 of the source operand must be zero.

ENQCMDS formats its source data differently from ENQCMD. Specifically, it formats them into **command data** as follows:

- Command[19:0] get bits 19:0 of the source operand that was read from memory. These 20 bits communicate a process address-space identifier (PASID).
- Command[30:20] are zero.
- Command[511:31] get bits 511:31 of the source operand that was read from memory. Bit 31 communicates a privilege identification (0 = user; 1 = supervisor).

The ENQCMDS instruction then uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).¹ The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store’s command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMDS instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device’s enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons.

1. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

The ENQCMDS instruction may be executed only if CPL = 0. Availability of the ENQCMDS instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.(EAX=07H, ECX=0H):ECX[bit 29]).

Operation

DEST := SRC;

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMDS int_enqcmds(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

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 destination linear address is not aligned to a 64-byte boundary. If the current privilege level is not 0. If bits 30:20 of the source operand are not all zero.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0. If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH. If destination linear address is not aligned to a 64-byte boundary. If bits 30:20 of the source operand are not all zero.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The ENQCMDS 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 non-canonical form.
#GP(0)	If the memory address is in non-canonical form. If destination linear address is not aligned to a 64-byte boundary. If the current privilege level is not 0. If bits 30:20 of the source operand are not all zero.
#PF(fault-code)	For a page fault.

#UD If CPUID.07H.0H:ECX.ENQCMD[bit 29].
 If the LOCK prefix is used.

EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	A	VV	SSE4_1	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	A	V/V	AVX	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	B	V/V	AVX512F	Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	N/A

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.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 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/EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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.

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

VEXTRACTPS (EVEX and 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 0FFFFFFFh

DEST[63:32] := 0

ELSE

DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFh

FI

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 0FFFFFFFh

DEST[63:32] := 0

ELSE

DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFh

FI

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS int _mm_extract_ps (__m128 a, const int idx);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-57, “Type E9NF Class Exception Conditions.”

Additionally:

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

F2XM1—Compute 2^x-1

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with $(2^{\text{ST}(0)} - 1)$.

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 3-16. 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:

$$x^y := 2^{(y * \log_2 x)}$$

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

Operation

$\text{ST}(0) := (2^{\text{ST}(0)} - 1);$

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 3-17. Results Obtained from FABS

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
$-F$	$+F$
-0	$+0$
$+0$	$+0$
$+F$	$+F$
$+\infty$	$+\infty$
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

$ST(0) := |ST(0)|;$

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 m32fp	Valid	Valid	Add m32fp to ST(0) and store result in ST(0).
DC /0	FADD m64fp	Valid	Valid	Add m64fp 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 m32int	Valid	Valid	Add m32int to ST(0) and store result in ST(0).
DE /0	FIADD m16int	Valid	Valid	Add m16int 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 3-18.

Table 3-18. FADD/FADDP/FIADD Results

SRC	DEST							
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$ or $-I$	$-\infty$	$-F$	SRC	SRC	$\pm F$ or ± 0	$+\infty$	NaN
	-0	$-\infty$	DEST	-0	± 0	DEST	$+\infty$	NaN
	$+0$	$-\infty$	DEST	± 0	$+0$	DEST	$+\infty$	NaN
	$+F$ or $+I$	$-\infty$	$\pm F$ or ± 0	SRC	SRC	$+F$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	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 + ConvertToDoubleExtendedPrecisionFP(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 m80bcd	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) := ConvertToDoubleExtendedPrecisionFP(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 3-19. 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 := 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.

FCFS—Change Sign

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E0	FCFS	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 3-20. FCFS Results

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
$-F$	$+F$
-0	$+0$
$+0$	-0
$+F$	$-F$
$+\infty$	$-\infty$
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

$\text{SignBit}(\text{ST}(0)) := \text{NOT}(\text{SignBit}(\text{ST}(0)))$;

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 ¹	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 ¹	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

NOTES:

1. 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 later Intel processors, except for the Intel Quark™ X1000 processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCSR 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 ¹	Instruction	64-Bit Mode	Compat/Leg Mode ¹	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	FCMOVNB 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:

1. 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);
FI;
```

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 m32fp	Valid	Valid	Compare ST(0) with m32fp.
DC /2	FCOM m64fp	Valid	Valid	Compare ST(0) with m64fp.
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 m32fp	Valid	Valid	Compare ST(0) with m32fp and pop register stack.
DC /3	FCOMP m64fp	Valid	Valid	Compare ST(0) with m64fp 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 3-21. 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 Floating-Point” 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 3-22. 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 Floating-Point” 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;
    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;
    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 approximate cosine.

Description

Computes the approximate 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 3-23. 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 . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” 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)| < 263
THEN
    C2 := 0;
    ST(0) := FCOS(ST(0)); // approximation of cosine
ELSE (* Source operand is out-of-range *)
    C2 := 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 m32fp	Valid	Valid	Divide ST(0) by m32fp and store result in ST(0).
DC /6	FDIV m64fp	Valid	Valid	Divide ST(0) by m64fp 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 m32int	Valid	Valid	Divide ST(0) by m32int and store result in ST(0).
DE /6	FIDIV m16int	Valid	Valid	Divide ST(0) by m16int 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 3-24. FDIV/FDIVP/FIDIV Results

		DEST						
SRC		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	$+0$	$+0$	-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. $\pm\infty / \pm\infty$; $\pm 0 / \pm 0$
#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 m32fp	Valid	Valid	Divide m32fp by ST(0) and store result in ST(0).
DC /7	FDIVR m64fp	Valid	Valid	Divide m64fp by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(0), ST(i)	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 m32int	Valid	Valid	Divide m32int by ST(0) and store result in ST(0).
DE /7	FIDIVR m16int	Valid	Valid	Divide m16int 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 3-25. FDIVR/FDIVRP/FIDIVR Results

SRC	DEST						
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$-\infty$	*	$+\infty$	$+\infty$	$-\infty$	$-\infty$	*
	$-F$	$+0$	$+F$	**	**	$-F$	-0
	$-I$	$+0$	$+F$	**	**	$-F$	-0
	-0	$+0$	$+0$	*	*	-0	-0
	$+0$	-0	-0	*	*	$+0$	$+0$
	$+I$	-0	$-F$	**	**	$+F$	$+0$
	$+F$	-0	$-F$	**	**	$+F$	$+0$
	$+\infty$	*	$-\infty$	$-\infty$	$+\infty$	$+\infty$	*
	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 := ConvertToDoubleExtendedPrecisionFP(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. $\pm\infty / \pm\infty$; $\pm 0 / \pm 0$
#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

TAG(i) := 11B;

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 m16int	Valid	Valid	Compare ST(0) with m16int.
DA /2	FICOM m32int	Valid	Valid	Compare ST(0) with m32int.
DE /3	FICOMP m16int	Valid	Valid	Compare ST(0) with m16int and pop stack register.
DA /3	FICOMP m32int	Valid	Valid	Compare ST(0) with m32int 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 3-26. 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 Floating-Point” 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 $-0.0 := +0.0$.

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;

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.

#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 m16int	Valid	Valid	Push m16int onto the FPU register stack.
DB /0	FILD m32int	Valid	Valid	Push m32int onto the FPU register stack.
DF /5	FILD m64int	Valid	Valid	Push m64int 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) := ConvertToDoubleExtendedPrecisionFP(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 ¹	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

NOTES:

1. 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 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 later Intel processors, except for the Intel Quark™ X1000 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 m16int	Valid	Valid	Store ST(0) in m16int.
DB /2	FIST m32int	Valid	Valid	Store ST(0) in m32int.
DF /3	FISTP m16int	Valid	Valid	Store ST(0) in m16int and pop register stack.
DB /3	FISTP m32int	Valid	Valid	Store ST(0) in m32int and pop register stack.
DF /7	FISTP m64int	Valid	Valid	Store ST(0) in m64int 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 3-27. FIST/FISTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < -0$	**
-0	0
$+0$	0
$+0 < F < +1$	**
$F \geq +1$	$+I$
$+\infty$ 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.
 ** 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. Set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format. Source operand is an SNaN, QNaN, $\pm\infty$, or unsupported 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.

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 3-28. 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 := 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.
#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.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#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.
#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 m32fp	Valid	Valid	Push m32fp onto the FPU register stack.
DD /0	FLD m64fp	Valid	Valid	Push m64fp onto the FPU register stack.
DB /5	FLD m80fp	Valid	Valid	Push m80fp 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.

NOTE

When the FLD instruction loads a denormal value and the DM bit in the CW is not masked, an exception is flagged but the value is still pushed onto the x87 stack.

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);
FI;
TOP := TOP - 1;
IF SRC is memory-operand
    THEN
        ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) := temp;
FI;
```

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 “Approximation of 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 m2byte.

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 unmask 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 m14/28byte	Valid	Valid	Load FPU environment from m14byte or m28byte.

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 m32fp	Valid	Valid	Multiply ST(0) by m32fp and store result in ST(0).
DC /1	FMUL m64fp	Valid	Valid	Multiply ST(0) by m64fp 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 m32int	Valid	Valid	Multiply ST(0) by m32int and store result in ST(0).
DE /1	FIMUL m16int	Valid	Valid	Multiply ST(0) by m16int 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 3-29. FMUL/FMULP/FIMUL Results

SRC	DEST						
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$-\infty$	$+\infty$	$+\infty$	*	*	$-\infty$	$-\infty$
	$-F$	$+\infty$	$+F$	$+0$	-0	$-F$	$-\infty$
	$-I$	$+\infty$	$+F$	$+0$	-0	$-F$	$-\infty$
	-0	*	$+0$	$+0$	-0	-0	*
	$+0$	*	-0	-0	$+0$	$+0$	*
	$+I$	$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$+F$	$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$
	$+\infty$	$-\infty$	$-\infty$	*	*	$+\infty$	$+\infty$
	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 * ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST := DEST * SRC;

FI;

IF Instruction = FMULP

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.
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 and the FPU Instruction Pointer.

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 ¹	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F3	FPATAN	Valid	Valid	Replace ST(1) with $\arctan(\text{ST}(1)/\text{ST}(0))$ and pop the register stack.

NOTES:

1. 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 3-30. FPATAN Results

		ST(0)						
ST(1)		$-\infty$	− F	− 0	+ 0	+ F	$+\infty$	NaN
	$-\infty$	$-3\pi/4^*$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4^*$	NaN
	− F	− p	$-\pi$ to $-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$ to -0	− 0	NaN
	− 0	− p	− p	− p [*]	− 0 [*]	− 0	− 0	NaN
	+ 0	+ p	+ p	+ π^*	+ 0 [*]	+ 0	+ 0	NaN
	+ F	+ p	+ π to $+\pi/2$	+ $\pi/2$	+ $\pi/2$	+ $\pi/2$ 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:

$$0 \leq |\text{ST}(1)| < |\text{ST}(0)| < +\infty$$

Operation

$ST(1) := \arctan(ST(1) / ST(0));$
 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:

$$\text{Remainder} := \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of $[\text{ST}(0) / \text{ST}(1)]$ 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 3-31. FPREM Results

		ST(1)						
		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
ST(0)	$-\infty$	*	*	*	*	*	*	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
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) 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:

$$\text{Remainder} := \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of $[\text{ST}(0) / \text{ST}(1)]$ 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 3-32. FPREM1 Results

		ST(1)						
ST(0)		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	*	*	*	*	*	NaN
	$-F$	ST(0)	$\pm F$ or -0	*	*	$\pm F$ or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	$+0$	$+0$	$+0$	*	*	$+0$	+0	NaN
	$+F$	ST(0)	$\pm F$ or $+0$	*	*	$\pm F$ or $+0$	ST(0)	NaN
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) 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 approximate tangent and push 1 onto the FPU stack.

Description

Computes the approximate 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 3-33. 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 . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" 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) < 263
  THEN
    C2 := 0;
    ST(0) := fptan(ST(0)); // approximation of tan
    TOP := TOP – 1;
    ST(0) := 1.0;
  ELSE (* Source operand is out-of-range *)
    C2 := 1;
FI;

```

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) := RoundToIntegerValue(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 m94/108byte	Valid	Valid	Load FPU state from m94byte or m108byte.

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 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 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, 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.
#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 m94/108byte	Valid	Valid	Store FPU state to m94byte or m108byte after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE ¹ m94/108byte	Valid	Valid	Store FPU environment to m94byte or m108byte without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

NOTES:

1. 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 FPU's 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 later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

(* Save FPU State and Registers *)

```
DEST[FPUControlWord] := FPUControlWord;
DEST[FPUStatusWord] := FPUStatusWord;
DEST[FPUTagWord] := FPUTagWord;
DEST[FPUDataPointer] := FPUDataPointer;
DEST[FPUInstructionPointer] := FPUInstructionPointer;
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;
FPUInstructionPointer := 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)	<p>If destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#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 3-34. FSCALE Results

ST(0)	ST(1)							
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	NaN	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	-0	$-F$	$-F$	$-F$	$-F$	$-\infty$	NaN
	-0	-0	-0	-0	-0	-0	NaN	NaN
	$+0$	$+0$	$+0$	$+0$	$+0$	$+0$	NaN	NaN
	$+F$	$+0$	$+F$	$+F$	$+F$	$+F$	$+\infty$	NaN
	$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	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

$$ST(0) := ST(0) * 2^{\text{RoundTowardZero}(ST(1))}$$

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 the approximate of its sine.

Description

Computes an approximation of 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 3-35. FSIN Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	-1 to $+1$
-0	-0
$+0$	$+0$
$+F$	-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 . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/4$. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” 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 := 0;
    ST(0) := fsin(ST(0)); // approximation of the mathematical sin function
  ELSE (* Source operand out of range *)
    C2 := 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 approximate sine, and push the approximate cosine onto the register stack.

Description

Computes both the approximate 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 3-36. FSINCOS Results

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
$-F$	-1 to $+1$	-1 to $+1$
-0	$+1$	-0
$+0$	$+1$	$+0$
$+F$	-1 to $+1$	-1 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 . However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" 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) < 263
    THEN
        C2 := 0;
        TEMP := fcos(ST(0)); // approximation of cosine
        ST(0) := fsin(ST(0)); // approximation of sine
        TOP := TOP – 1;
        ST(0) := TEMP;
    ELSE (* Source operand out of range *)
        C2 := 1;
FI;

```

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 3-37. 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) := 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 m32fp	Valid	Valid	Copy ST(0) to m32fp.
DD /2	FST m64fp	Valid	Valid	Copy ST(0) to m64fp.
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP m32fp	Valid	Valid	Copy ST(0) to m32fp and pop register stack.
DD /3	FSTP m64fp	Valid	Valid	Copy ST(0) to m64fp and pop register stack.
DB /7	FSTP m80fp	Valid	Valid	Copy ST(0) to m80fp 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 m2byte	Valid	Valid	Store FPU control word to m2byte after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW ¹ m2byte	Valid	Valid	Store FPU control word to m2byte without checking for pending unmasked floating-point exceptions.

NOTES:

1. 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 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 later Intel processors, except for the Intel Quark™ X1000 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)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#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 m14/28byte	Valid	Valid	Store FPU environment to m14byte or m28byte after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV ¹ m14/28byte	Valid	Valid	Store FPU environment to m14byte or m28byte without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

NOTES:

1. 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 later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST[FPUControlWord] := FPUControlWord;
DEST[FPUStatusWord] := FPUStatusWord;
DEST[FPUTagWord] := FPUTagWord;
DEST[FPUDataPointer] := FPUDataPointer;
DEST[FPUInstructionPointer] := FPUInstructionPointer;
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 m2byte	Valid	Valid	Store FPU status word at m2byte 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 ¹ m2byte	Valid	Valid	Store FPU status word at m2byte without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW ¹ AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

NOTES:

1. 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 later Intel processors, except for the Intel Quark™ X1000 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 m32fp	Valid	Valid	Subtract m32fp from ST(0) and store result in ST(0).
DC /4	FSUB m64fp	Valid	Valid	Subtract m64fp 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 m32int	Valid	Valid	Subtract m32int from ST(0) and store result in ST(0).
DE /4	FISUB m16int	Valid	Valid	Subtract m16int 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 3-38 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 3-38. FSUB/FSUBP/FISUB Results

DEST	SRC							
		$-\infty$	$-F$ or $-I$	-0	$+0$	$+F$ or $+I$	$+\infty$	NaN
	$-\infty$	*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	$+\infty$	$\pm F$ or ± 0	DEST	DEST	$-F$	$-\infty$	NaN
	-0	$+\infty$	$-SRC$	± 0	-0	$-SRC$	$-\infty$	NaN
	$+0$	$+\infty$	$-SRC$	$+0$	± 0	$-SRC$	$-\infty$	NaN
	$+F$	$+\infty$	$+F$	DEST	DEST	$\pm F$ or ± 0	$-\infty$	NaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	*	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 – ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST := DEST – SRC;

FI;

IF Instruction = FSUBP

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 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 m32fp	Valid	Valid	Subtract ST(0) from m32fp and store result in ST(0).
DC /5	FSUBR m64fp	Valid	Valid	Subtract ST(0) from m64fp 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 m32int	Valid	Valid	Subtract ST(0) from m32int and store result in ST(0).
DE /5	FISUBR m16int	Valid	Valid	Subtract ST(0) from m16int 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 ($SRC - DEST = \text{result}$).

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 3-39. FSUBR/FSUBRP/FISUBR Results

DEST	SRC							
		$-\infty$	$-F$ or $-I$	-0	$+0$	$+F$ or $+I$	$+\infty$	NaN
	$-\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	$-F$	$-\infty$	$\pm F$ or ± 0	$-\text{DEST}$	$-\text{DEST}$	$+F$	$+\infty$	NaN
	-0	$-\infty$	SRC	± 0	$+0$	SRC	$+\infty$	NaN
	$+0$	$-\infty$	SRC	-0	± 0	SRC	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	$-\text{DEST}$	$-\text{DEST}$	$\pm F$ or ± 0	$+\infty$	NaN
	$+\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	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 := ConvertToDoubleExtendedPrecisionFP(SRC) – DEST;

ELSE (* Source operand is floating-point value *)

DEST := SRC – DEST; FI;

IF Instruction = FSUBRP

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 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 3-40. 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 Floating-Point” 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 $(-0.0 := +0.0)$.

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 := 111;

ST(0) > 0.0: C3, C2, C0 := 000;

ST(0) < 0.0: C3, C2, C0 := 001;

ST(0) = 0.0: C3, C2, C0 := 100;

ESAC;

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 See Table 3-40.

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 3-41. 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 Floating-Point” 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;

FI;

FI;

IF Instruction = FUCOMP

THEN

PopRegisterStack;

FI;

IF Instruction = FUCOMPP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

C0, C2, C3 See Table 3-41.

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 Floating-Point

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 3-42. 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 := 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 3-42.

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.
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
NP OF AE /1 FXRSTOR m512byte	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from m512byte.
NP REX.W + OF AE /1 FXRSTOR64 m512byte	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from m512byte.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

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 3-43 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 3-46 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-47 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 3-43, Table 3-46, or Table 3-47. 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

IF 64-Bit Mode

THEN

(x87 FPU, MMX, XMM15-XMM0, MXCSR) Load(SRC);

ELSE

(x87 FPU, MMX, XMM7-XMM0, MXCSR) := Load(SRC);

FI;

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
NP OF AE /0 FXSAVE m512byte	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to m512byte.
NP REX.W + OF AE /0 FXSAVE64 m512byte	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to m512byte.

Instruction Operand Encoding

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

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 3-43 shows the layout of the state information in memory when the processor is operating in legacy modes.

Table 3-43. 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		FCS		FIP[31:0]				FOP		Rsvd	FTW	FSW		FCW		0
MXCSR_MASK				MXCSR				Rsrvd		FDS		FDP[31:0]				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
Reserved																288

Table 3-43. 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																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 3-43 are defined in Table 3-44.

Table 3-44. Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, 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 Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the layout of the x87 FPU opcode field.
FIP	x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-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. 64-bit mode with REX.W — 64-bit IP offset. 64-bit mode without REX.W — 32-bit IP offset. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a description of the x87 FPU instruction pointer.
FCS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.

Table 3-44. Field Definitions (Contd.)

Field	Definition
FDP	x87 FPU Instruction Operand (Data) Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-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. 64-bit mode with REX.W — 64-bit DP offset. 64-bit mode without REX.W — 32-bit DP offset. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of the x87 FPU operand pointer.
FDS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, 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 Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, 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 a 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 floating-point data (assuming the stored data was not the contents of MMX technology registers) using Table 3-45.

Table 3-45. 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 3-46), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-47), the FPU IP and FPU DP pointers are 32-bits.

Table 3-46. 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		
FIP								FOP	Reserved	FTW	FSW	FCW	0				
MXCSR_MASK				MXCSR				FDP								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	
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 3-47. Layout of the 64-Bit Mode FXSAVE Map (RFX.W = 0)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved	FCS	FIP[31:0]		FOP	Reserved	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Reserved	FDS		FDP[31:0]		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
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, XMM15-XMM0,
        MXCSR);
      ELSE
        DEST := Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-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 $\log_b(x)$ 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 3-48. FYL2X Results

		ST(0)							
ST(1)		$-\infty$	$-F$	± 0	$+0 < +F < +1$	$+1$	$+F > +1$	$+\infty$	NaN
	$-\infty$	*	*	$+\infty$	$+\infty$	*	$-\infty$	$-\infty$	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):

$$\log_b x := (\log_2 b)^{-1} * \log_2 x$$

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

Operation

$ST(1) := ST(1) * \log_2 ST(0);$

PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	Set if result was rounded up; cleared otherwise.
	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 3-49. FYL2XP1 Results

		ST(0)				
ST(1)		$-(1 - (\sqrt{2}/2)) \text{ to } -0$	-0	+0	$+0 \text{ to } +(1 - (\sqrt{2}/2))$	NaN
	$-\infty$	$+\infty$	*	*	$-\infty$	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
	$+\infty$	$-\infty$	*	*	$+\infty$	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:

$$\text{scale factor} := \log_n 2$$

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

Operation

$ST(1) := ST(1) * \log_2(ST(0) + 1.0);$

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.

GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes inverse affine transformation in the finite field $GF(2^8)$.
VEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field $GF(2^8)$.
VEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field $GF(2^8)$.
EVEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes inverse affine transformation in the finite field $GF(2^8)$.
EVEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes inverse affine transformation in the finite field $GF(2^8)$.
EVEX.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F GFNI	Computes inverse affine transformation in the finite field $GF(2^8)$.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEINVB instruction computes an affine transformation in the Galois Field 2^8 . For this instruction, an affine transformation is defined by $A * \text{inv}(x) + b$ where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.

Table 3-50. Inverse Byte Listings

-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	9
5	ED	5C	5	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	6	A1	FA	81	82
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	2	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	B	28	2F	A3	DA	D4	E4	F	A9	27	53	4	1B	FC	AC	E6
D	7A	7	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	D	D6	EB	C6	E	CF	AD	8	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	3	8C	DD	9C	7D	A0	CD	1A	41	1C

Operation

```

define affine_inverse_byte(tsrc2qw, src1byte, imm):
  FOR i := 0 to 7:
    * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
    * inverse(x) is defined in the table above *
    retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
  return retbyte

```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX Encoded Version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1:

IF SRC2 is memory and EVEX.b==1:

tsrc2 := SRC2.qword[0]

ELSE:

tsrc2 := SRC2.qword[j]

FOR b := 0 to 7:

IF k1[j*8+b] OR *no writemask*:

FOR i := 0 to 7:

DEST.qword[j].byte[b] := affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)

ELSE IF *zeroing*:

DEST.qword[j].byte[b] := 0

ELSE DEST.qword[j].byte[b] remains unchanged

DEST[MAX_VL-1:VL] := 0

VGFP8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256)

FOR j := 0 TO KL-1:

FOR b := 0 to 7:

DEST.qword[j].byte[b] := affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)

DEST[MAX_VL-1:VL] := 0

GF2P8AFFINEINVQB srcdest, src1, imm8 (128b SSE Encoded Version)

FOR j := 0 TO 1:

FOR b := 0 to 7:

SRCDEST.qword[j].byte[b] := affine_inverse_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8AFFINEINVQB __m128i __mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);

(V)GF2P8AFFINEINVQB __m128i __mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);

(V)GF2P8AFFINEINVQB __m128i __mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);

VGFP8AFFINEINVQB __m256i __mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, int);

VGFP8AFFINEINVQB __m256i __mm256_mask_gf2p8affineinv_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);

VGFP8AFFINEINVQB __m256i __mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, int);

VGFP8AFFINEINVQB __m512i __mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, int);

VGFP8AFFINEINVQB __m512i __mm512_mask_gf2p8affineinv_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);

VGFP8AFFINEINVQB __m512i __mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."

GF2P8AFFINEQB—Galois Field Affine Transformation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CE /r /ib GF2P8AFFINEQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes affine transformation in the finite field $GF(2^8)$.
VEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field $GF(2^8)$.
VEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field $GF(2^8)$.
EVEX.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes affine transformation in the finite field $GF(2^8)$.
EVEX.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes affine transformation in the finite field $GF(2^8)$.
EVEX.512.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F GFNI	Computes affine transformation in the finite field $GF(2^8)$.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEQB instruction computes an affine transformation in the Galois Field 2^8 . For this instruction, an affine transformation is defined by $A * x + b$ where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

Operation

define parity(x):

```

    t := 0           // single bit
    FOR i := 0 to 7:
        t = t xor x.bit[i]
    return t

```

define affine_byte(tsrc2qw, src1byte, imm):

```

    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte

```

VGFP8AFFINEQB dest, src1, src2, imm8 (EVEX Encoded Version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1:

IF SRC2 is memory and EVEX.b==1:

tsrc2 := SRC2.qword[0]

ELSE:

tsrc2 := SRC2.qword[j]

FOR b := 0 to 7:

IF k1[j*8+b] OR *no writemask*:

DEST.qword[j].byte[b] := affine_byte(tsrc2, SRC1.qword[j].byte[b], imm8)

ELSE IF *zeroing*:

DEST.qword[j].byte[b] := 0

ELSE DEST.qword[j].byte[b] remains unchanged

DEST[MAX_VL-1:VL] := 0

VGFP8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256)

FOR j := 0 TO KL-1:

FOR b := 0 to 7:

DEST.qword[j].byte[b] := affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)

DEST[MAX_VL-1:VL] := 0

GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE Encoded Version)

FOR j := 0 TO 1:

FOR b := 0 to 7:

SRCDEST.qword[j].byte[b] := affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8AFFINEQB __m128i __mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);

(V)GF2P8AFFINEQB __m128i __mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);

(V)GF2P8AFFINEQB __m128i __mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);

VGFP8AFFINEQB __m256i __mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);

VGFP8AFFINEQB __m256i __mm256_mask_gf2p8affine_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);

VGFP8AFFINEQB __m256i __mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);

VGFP8AFFINEQB __m512i __mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);

VGFP8AFFINEQB __m512i __mm512_mask_gf2p8affine_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);

VGFP8AFFINEQB __m512i __mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."

GF2P8MULB—Galois Field Multiply Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F38 CF /r GF2P8MULB xmm1, xmm2/m128	A	V/V	GFNI	Multiplies elements in the finite field $GF(2^8)$.
VEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1, xmm2, xmm3/m128	B	V/V	AVX GFNI	Multiplies elements in the finite field $GF(2^8)$.
VEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1, ymm2, ymm3/m256	B	V/V	AVX GFNI	Multiplies elements in the finite field $GF(2^8)$.
EVEX.128.66.0F38.W0 CF /r VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL GFNI	Multiplies elements in the finite field $GF(2^8)$.
EVEX.256.66.0F38.W0 CF /r VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL GFNI	Multiplies elements in the finite field $GF(2^8)$.
EVEX.512.66.0F38.W0 CF /r VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512F GFNI	Multiplies elements in the finite field $GF(2^8)$.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

The instruction multiplies elements in the finite field $GF(2^8)$, operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field $GF(2^8)$ is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

Operation

```
define gf2p8mul_byte(src1byte, src2byte):
    tword := 0
    FOR i := 0 to 7:
        IF src2byte.bit[i]:
            tword := tword XOR (src1byte << i)
        * carry out polynomial reduction by the characteristic polynomial p*
    FOR i := 14 downto 8:
        p := 0x11B << (i-8)      *0x11B = 0000_0001_0001_1011 in binary*
        IF tword.bit[i]:
            tword := tword XOR p
    return tword.byte[0]
```

VGF2P8MULB dest, src1, src2 (EVEX Encoded Version)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1:

IF k1[j] OR *no writemask*:

DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])

ELSE IF *zeroing*:

DEST.byte[j] := 0

* ELSE DEST.byte[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

VGF2P8MULB dest, src1, src2 (128b and 256b VEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256)

FOR j := 0 TO KL-1:

DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])

DEST[MAX_VL-1:VL] := 0

GF2P8MULB srcdest, src1 (128b SSE Encoded Version)

FOR j := 0 TO 15:

SRCDDEST.byte[j] := gf2p8mul_byte(SRCDDEST.byte[j], SRC1.byte[j])

Intel C/C++ Compiler Intrinsic Equivalent

(V)GF2P8MULB __m128i __mm_gf2p8mul_epi8(__m128i, __m128i);

(V)GF2P8MULB __m128i __mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);

(V)GF2P8MULB __m128i __mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);

VGF2P8MULB __m256i __mm256_gf2p8mul_epi8(__m256i, __m256i);

VGF2P8MULB __m256i __mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);

VGF2P8MULB __m256i __mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);

VGF2P8MULB __m512i __mm512_gf2p8mul_epi8(__m512i, __m512i);

VGF2P8MULB __m512i __mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);

VGF2P8MULB __m512i __mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded: See Table 2-49, "Type E4 Class Exception Conditions."

HADDPD—Packed Double Precision Floating-Point Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal add packed double precision floating-point values from xmm2/m128 to xmm1.
VEX.128.66.0F.WIG 7C /r VHADDPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal add packed double precision floating-point values from xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 7C /r VHADDPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal add packed double precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

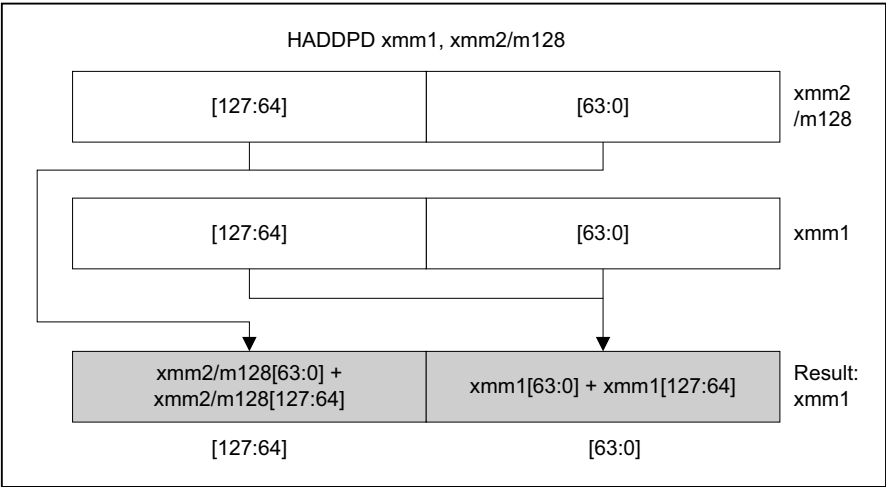
Description

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 3-17 for HADDPD; see Figure 3-18 for VHADDPD.



OM15993

Figure 3-17. HADDPD—Packed Double Precision Floating-Point Horizontal Add

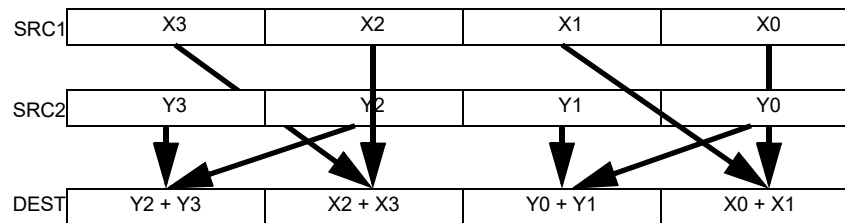


Figure 3-18. VHADDPD Operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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 Table 2-19, “Type 2 Class Exception Conditions.”

HADDPS—Packed Single Precision Floating-Point Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7C /r HADDPS xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal add packed single precision floating-point values from xmm2/m128 to xmm1.
VEX.128.F2.0F.WIG 7C /r VHADDPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal add packed single precision floating-point values from xmm2 and xmm3/mem.
VEX.256.F2.0F.WIG 7C /r VHADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal add packed single precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

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 3-19 for HADDPS; see Figure 3-20 for VHADDPS.

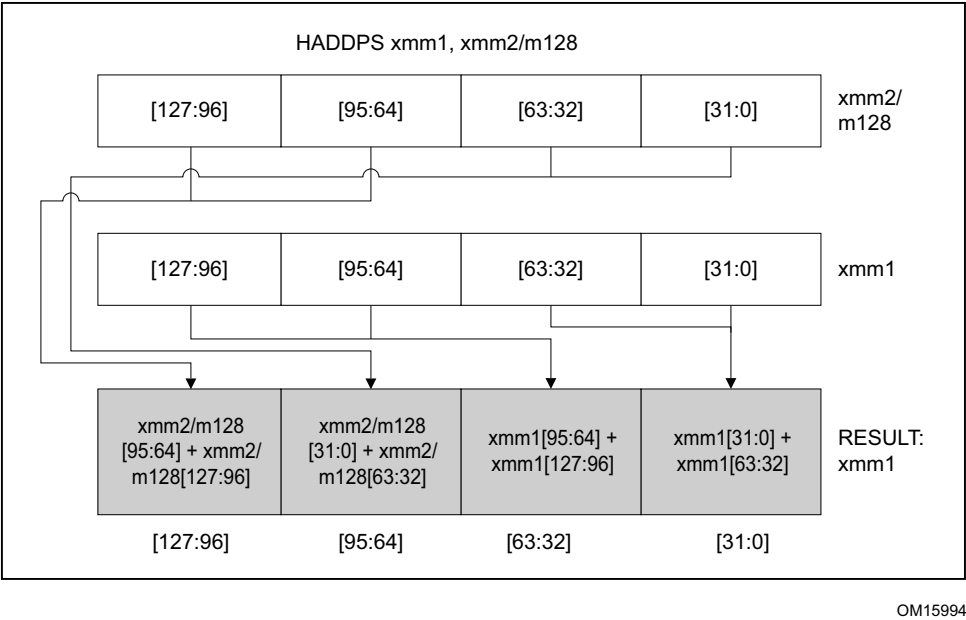


Figure 3-19. HADDPS—Packed Single Precision Floating-Point Horizontal Add

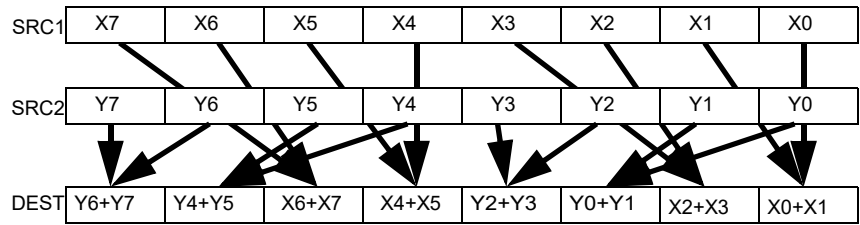


Figure 3-20. VHADDPS Operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-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 (MAXVL-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)

$\text{DEST}[31:0] := \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] := \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] := \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] := \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[\text{MAXVL}-1:128]$ (Unmodified)

VHADDPS (VEX.128 Encoded Version)

$\text{DEST}[31:0] := \text{SRC1}[63:32] + \text{SRC1}[31:0]$
 $\text{DEST}[63:32] := \text{SRC1}[127:96] + \text{SRC1}[95:64]$
 $\text{DEST}[95:64] := \text{SRC2}[63:32] + \text{SRC2}[31:0]$
 $\text{DEST}[127:96] := \text{SRC2}[127:96] + \text{SRC2}[95:64]$
 $\text{DEST}[\text{MAXVL}-1:128] := 0$

VHADDPS (VEX.256 Encoded Version)

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

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 Table 2-19, “Type 2 Class Exception Conditions.”

HLT—Halt

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F4	HLT	Z0	Valid	Valid	Halt

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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.

HRESET—History Reset

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 3A F0 C0 /ib HRESET imm8, <EAX>	A	V/V	HRESET	Processor history reset request. Controlled by the EAX implicit operand.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

Requests the processor to selectively reset selected components of hardware history maintained by the current logical processor. HRESET operation is controlled by the implicit EAX operand. The value of the explicit imm8 operand is ignored. This instruction can only be executed at privilege level 0.

The HRESET instruction can be used to request reset of multiple components of hardware history. Prior to the execution of HRESET, the system software must take the following steps:

1. Enumerate the HRESET capabilities via CPUID.20H.0H:EBX, which indicates what components of hardware history can be reset.
2. Only the bits enumerated by CPUID.20H.0H:EBX can be set in the IA32_HRESET_ENABLE MSR.

HRESET causes a general-protection exception (#GP) if EAX sets any bits that are not set in the IA32_HRESET_ENABLE MSR.

Any attempt to execute the HRESET instruction inside a transactional region will result in a transaction abort.

Operation

```
IF EAX = 0
  THEN NOP
  ELSE
    FOREACH i such that EAX[i] = 1
      Reset prediction history for feature i
    FI
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If CPL > 0 or (EAX AND NOT IA32_HRESET_ENABLE) ≠ 0.
 #UD If CPUID.07H.01H:EAX.HRESET[bit 22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

#GP(0) HRESET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal subtract packed double precision floating-point values from xmm2/m128 to xmm1.
VEX.128.66.0F.WIG 7D /r VHSUBPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal subtract packed double precision floating-point values from xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 7D /r VHSUBPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal subtract packed double precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

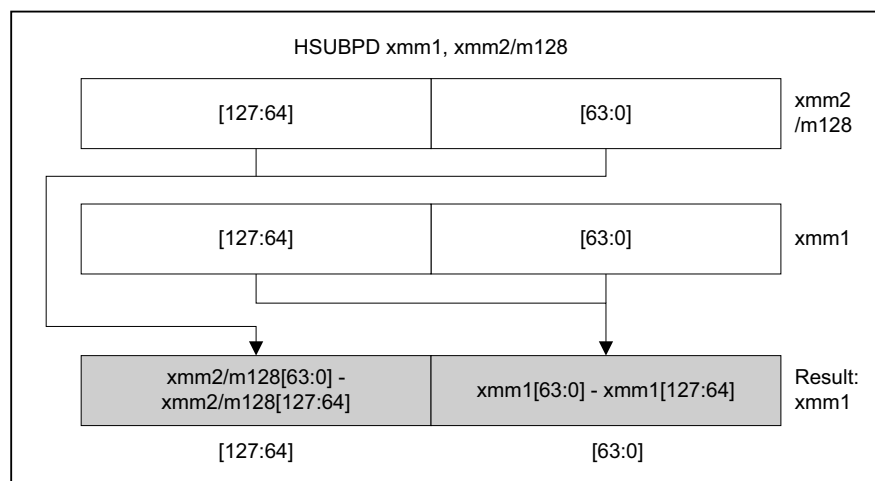
The HSUBPD instruction subtracts horizontally the packed double precision floating-point numbers of both operands.

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 3-21 for HSUBPD; see Figure 3-22 for VHSUBPD.



OM15995

Figure 3-21. HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract

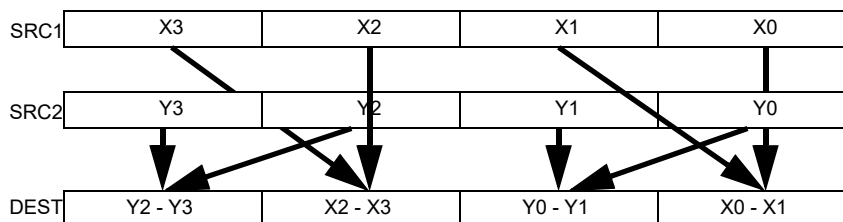


Figure 3-22. VHSUBPD operation

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 (MAXVL-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 (MAXVL-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[MAXVL-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[MAXVL-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 Table 2-19, “Type 2 Class Exception Conditions.”

HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7D /r HSUBPS xmm1, xmm2/m128	RM	V/V	SSE3	Horizontal subtract packed single precision floating-point values from xmm2/m128 to xmm1.
VEX.128.F2.0F.WIG 7D /r VHSUBPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal subtract packed single precision floating-point values from xmm2 and xmm3/mem.
VEX.256.F2.0F.WIG 7D /r VHSUBPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal subtract packed single precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

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 3-23 for HSUBPS; see Figure 3-24 for VHSUBPS.

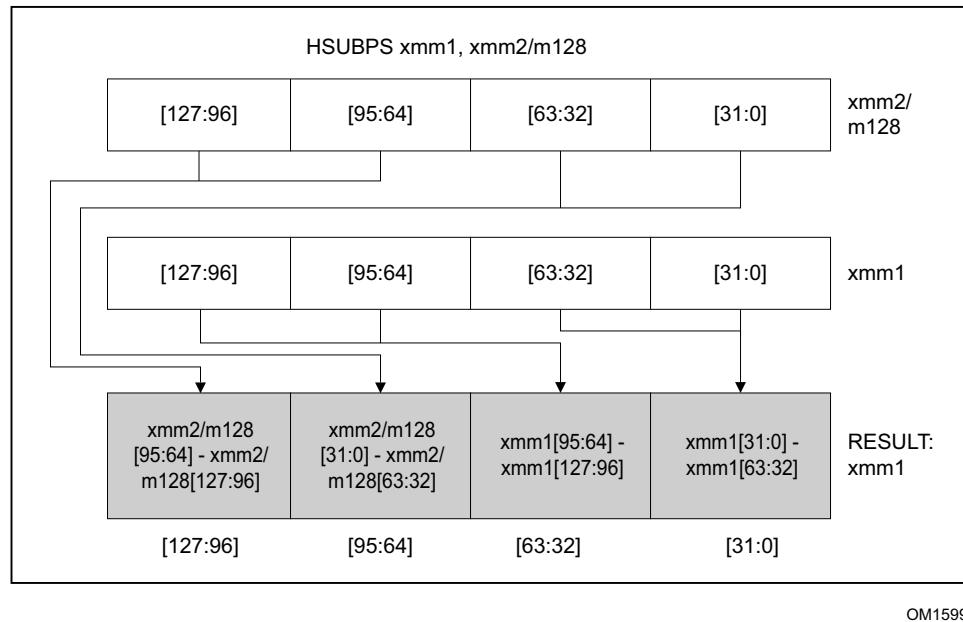


Figure 3-23. HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract

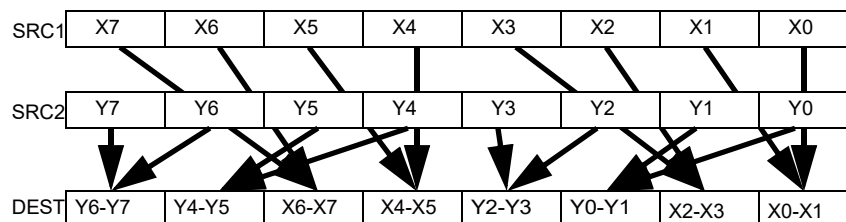


Figure 3-24. VHSUBPS Operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-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 (MAXVL-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)

$\text{DEST}[31:0] := \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] := \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] := \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] := \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[\text{MAXVL}-1:128]$ (Unmodified)

VHSUBPS (VEX.128 Encoded Version)

$\text{DEST}[31:0] := \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] := \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] := \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] := \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[\text{MAXVL}-1:128] := 0$

VHSUBPS (VEX.256 Encoded Version)

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

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 Table 2-19, “Type 2 Class Exception Conditions.”

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 := Quotient, AH := Remainder.
REX + F6 /7	IDIV r/m8 ¹	M	Valid	N.E.	Signed divide AX by r/m8, with result stored in AL := Quotient, AH := Remainder.
F7 /7	IDIV r/m16	M	Valid	Valid	Signed divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.
F7 /7	IDIV r/m32	M	Valid	Valid	Signed divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.
REX.W + F7 /7	IDIV r/m64	M	Valid	N.E.	Signed divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.

NOTES:

1. 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)	N/A	N/A	N/A

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 3-51.

Table 3-51. 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 ³¹ – 1
Doublequadword/quadword	RDX:RAX	r/m64	RAX	RDX	–2 ⁶³ to 2 ⁶³ – 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 > 7FFFH) or (temp < 8000H)
                (* If a positive result is greater than 7FFFH
                or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX := temp;
                    DX := DX:AX SignedModulus SRC;
            FI;
        ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
            THEN
                temp := EDX:EAX / SRC; (* Signed division *)
                IF (temp > 7FFFFFFFFFH) or (temp < 80000000H)
                    (* If a positive result is greater than 7FFFFFFFFFH
                    or a negative result is less than 80000000H *)
                    THEN
                        #DE; (* Divide error *)
                    ELSE
                        EAX := temp;
                        EDX := EDX:EAX SignedModulus SRC;
                FI;
            ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
                THEN
                    temp := RDX:RAX / SRC; (* Signed division *)
                    IF (temp > 7FFFFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
                        (* If a positive result is greater than 7FFFFFFFFFFFFFFFFFH
                        or a negative result is less than 8000000000000000H *)
                        THEN
                            #DE; (* Divide error *)
                        ELSE
                            RAX := temp;
                            RDX := RDX:RAX SignedModulus SRC;
                    FI;
                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 ¹	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.
0F AF /r	IMUL r16, r/m16	RM	Valid	Valid	word register := word register * r/m16.
0F AF /r	IMUL r32, r/m32	RM	Valid	Valid	doubleword register := doubleword register * r/m32.
REX.W + 0F 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:

1. 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)	N/A	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	N/A

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 (twice the size of the input operand) 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 intermediate product (twice the size of the input operand) is truncated and 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 intermediate product (twice the size of the first source operand) is truncated and 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 the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

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
      TMP_XP := AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
      AX := TMP_XP[15:0];
      IF SignExtend(TMP_XP[7:0]) = TMP_XP
        THEN CF := 0; OF := 0;
        ELSE CF := 1; OF := 1; FI;
    ELSE IF OperandSize = 16
      THEN
        TMP_XP := AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DX:AX := TMP_XP[31:0];
        IF SignExtend(TMP_XP[15:0]) = TMP_XP
          THEN CF := 0; OF := 0;
          ELSE CF := 1; OF := 1; FI;
    ELSE IF OperandSize = 32
      THEN
        TMP_XP := EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        EDX:EAX := TMP_XP[63:0];
        IF SignExtend(TMP_XP[31:0]) = TMP_XP
          THEN CF := 0; OF := 0;
          ELSE CF := 1; OF := 1; FI;
    ELSE (* OperandSize = 64 *)
      TMP_XP := RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
      EDX:EAX := TMP_XP[127:0];
      IF SignExtend(TMP_XP[63:0]) = TMP_XP
        THEN CF := 0; OF := 0;
        ELSE CF := 1; OF := 1; FI;
    FI;
  FI;

```

```

ELSE IF (NumberOfOperands = 2)
    THEN
        TMP_XP := DEST * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DEST := TruncateToOperandSize(TMP_XP);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF := 1; OF := 1;
            ELSE CF := 0; OF := 0; FI;
    ELSE (* NumberOfOperands = 3 *)
        TMP_XP := SRC1 * SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
        DEST := TruncateToOperandSize(TMP_XP);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF := 1; OF := 1;
            ELSE CF := 0; OF := 0; FI;
    FI;
FI;

```

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 ib	IN AL, imm8	I	Valid	Valid	Input byte from imm8 I/O port address into AL.
E5 ib	IN AX, imm8	I	Valid	Valid	Input word from imm8 I/O port address into AX.
E5 ib	IN EAX, imm8	I	Valid	Valid	Input dword from imm8 I/O port address into EAX.
EC	IN AL, DX	ZO	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX, DX	ZO	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX, DX	ZO	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	N/A	N/A	N/A
ZO	N/A	N/A	N/A	N/A

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 19, "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.
#PF(fault-code)	If a page fault occurs.
#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.
#PF(fault-code)	If a page fault occurs.
#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 ¹	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 ²	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:

1. 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.
2. 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)	N/A	N/A	N/A
O	opcode + rd (r, w)	N/A	N/A	N/A

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;

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 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.

INCSSPD/INCSSPQ—Increment Shadow Stack Pointer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /05 INCSSPD r32	R	V/V	CET_SS	Increment SSP by 4 * r32[7:0].
F3 REX.W 0F AE /05 INCSSPQ r64	R	V/N.E.	CET_SS	Increment SSP by 8 * r64[7:0].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
R	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

This instruction can be used to increment the current shadow stack pointer by the operand size of the instruction times the unsigned 8-bit value specified by bits 7:0 in the source operand. The instruction performs a pop and discard of the first and last element on the shadow stack in the range specified by the unsigned 8-bit value in bits 7:0 of the source operand.

Operation

IF CPL = 3

IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
THEN #UD; FI;

ELSE

IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
THEN #UD; FI;

FI;

IF (operand size is 64-bit)

THEN

Range := R64[7:0];

shadow_stack_load 8 bytes from SSP;

IF Range > 0

THEN shadow_stack_load 8 bytes from SSP + 8 * (Range - 1);

FI;

SSP := SSP + Range * 8;

ELSE

Range := R32[7:0];

shadow_stack_load 4 bytes from SSP;

IF Range > 0

THEN shadow_stack_load 4 bytes from SSP + 4 * (Range - 1);

FI;

SSP := SSP + Range * 4;

FI;

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

INCSSPD void _incsspd(int);

INCSSPQ void _incsspq(int);

Protected Mode Exceptions

- #UD If the LOCK prefix is used.
 If CR4.CET = 0.
 IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
 IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
- #PF(fault-code) If a page fault occurs.

Real-Address Mode Exceptions

- #UD The INCSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

- #UD The INCSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
6C	INS m8, DX	ZO	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INS m16, DX	ZO	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INS m32, DX	ZO	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6C	INSB	ZO	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI. ¹
6D	INSW	ZO	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INSD	ZO	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹

NOTES:

1. In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

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 19, “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.

These instructions may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

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;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	<p>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.</p> <p>If the destination is located in a non-writable segment.</p> <p>If an illegal memory operand effective address in the ES segments is given.</p>
#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)	<p>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.</p> <p>If the memory address is in a non-canonical form.</p>
#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 Scalar Single Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Insert a single precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	C	V/V	AVX512F	Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(register source form)

Copy a single precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand 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 (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX 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 (MAXVL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation**VINSERTPS (VEX.128 and EVEX 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[MAXVL-1:128] := 0

```

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[MAXVL-1:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD If VEX.L = 0.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”

INT *n*/INT0/INT3/INT1—Call to Interrupt Procedure

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CC	INT3	Z0	Valid	Valid	Generate breakpoint trap.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Generate software interrupt with vector specified by immediate byte.
CE	INT0	Z0	Invalid	Valid	Generate overflow trap if overflow flag is 1.
F1	INT1	Z0	Valid	Valid	Generate debug trap.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A
I	<i>imm8</i>	N/A	N/A	N/A

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 a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors 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), exception 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 INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6.¹ Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. 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, INT3, and INT1 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.

Each of the INT *n*, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.

even if the CPL is greater than the DPL of descriptor 1 in the IDT. (This behavior supports the use of INT1 by hardware vendors performing hardware debug.)

The vector 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 3-52. 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	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	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.
- S/W Applies to INT *n*, INT3, and INTO, but not to INT1.

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.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 17, “Control-Flow Enforcement Technology (CET)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for CET details.

Instruction ordering. Instructions following an INT *n* may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the INT *n* have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible). This applies also to the INTO, INT3, and INT1 instructions, but not to executions of INTO when EFLAGS.OF = 0.

Operation

The following operational description applies not only to the INT *n*, INTO, INT3, or INT1 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 $(\text{num} \& \text{FCH}) \mid \text{ext}$; (2) if `idt` is 1, the error code is $(\text{num} \ll 3) \mid 2 \mid \text{ext}$.

In many cases, the pseudofunction `error_code` is invoked with a pseudovalue 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 (INT *n*, INT3, or INTO), EXT is 0; otherwise (including INT1), EXT is 1.

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT n *)
      ELSE
        IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
          THEN
            Consult bit n of the software interrupt redirection bit map in the TSS;
            IF bit n is clear
              THEN (* redirect interrupt to 8086 program interrupt handler *)
                Push EFLAGS[15:0];    (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                Push CS;
                Push IP;
                IF IOPL = 3
                  THEN IF := 0; (* Clear interrupt flag *)
                  ELSE VIF := 0; (* Clear virtual interrupt flag *)
                FI;
                TF := 0; (* Clear trap flag *)
                load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
              ELSE
                IF IOPL = 3
                  THEN GOTO PROTECTED-MODE;
                  ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
                FI;
            FI;
          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;
      FI;
    FI;
  FI;

```

```

        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; does not apply to INT1 *)
        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;
    FI;
    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 *)
    FI;
    IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
        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*)
            FI;
        FI;
    END;

```

```

        FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    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)); FI;
        (* 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(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(error_code(TSS selector,0,EXT)); FI;
        (* 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); FI;
            Push(error code);
        FI;
    IF EIP not within code segment limit
        THEN #GP(EXT); FI;
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); FI; (* 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)); FI;
        (* 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)); FI;
        (* 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)); FI;
        (* 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; FI;
        (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
    FI;
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 *)
        FI;
    FI;
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)); FI;
                        (* 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)); FI;
                        (* 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);
                FI;
            IF NewSS is NULL
                THEN #TS(EXT); FI;
            IF NewSS index is not within its descriptor-table limits
            or NewSS RPL ≠ new code-segment DPL
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* 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)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            IF NewSS is not present
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
        ELSE (* IA-32e mode *)

```

```

IF IDT-gate IST = 0
    THEN TSSstackAddress := (new code-segment DPL << 3) + 4;
    ELSE TSSstackAddress := (IDT gate IST << 3) + 28;
FI;
IF (TSSstackAddress + 7) > current 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);
NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
IF IDT-gate IST = 0
    THEN
        NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
    ELSE
        NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT-gate IST << 3)
        (* Check if shadow stacks are enabled at CPL 0 *)
        IF ShadowStackEnabled(CPL 0)
            THEN NewSSP := 8 bytes loaded from NewSSPAddress; FI;
FI;
FI;
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)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        FI
    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)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                ELSE (* 64-bit IDT gate*)
                    IF StackAddress is non-canonical
                        THEN #SS(EXT); FI; (* 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 limits
                    THEN #GP(EXT); FI; (* 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); FI; (* Error code contains NULL selector *)
                RSP := NewRSP & FFFFFFFF0H;
                SS := NewSS;
            FI;
        IF IDT gate is 32-bit
            THEN
                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;

FI;

IF ShadowStackEnabled(CPL) AND CPL = 3
THEN
    IF IA32_EFER.LMA = 0
    THEN IA32_PL3_SSP := SSP;
    ELSE (* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
        IA32_PL3_SSP := LA_adjust(SSP);
    FI;
FI;

FI;

CPL := new code-segment DPL;
CS(RPL) := CPL;
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := NewSSP
    IF SSP & 0x07 != 0
    THEN #GP(0); FI;
    (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
    IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
    #GP(0); FI;
    IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)

```

```

    THEN #GP(0); FI;
    expected_token_value = SSP          (* busy bit - bit position 0 - must be clear *)
    new_token_value = SSP | BUSY_BIT    (* Set the busy bit *)
    IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
        THEN #GP(0); FI;
    IF oldSS.DPL != 3
        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit RIP*)
        ShadowStackPush8B(oldSSP);
    FI;
FI;
IF EndbranchEnabled (CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
    IA32_S_CET.SUPPRESS = 0
FI;
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)); FI;
                (* 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);
            FI;
        IF NewSS is NULL
            THEN #TS(EXT); FI; (* 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)); FI;
            (* 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)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF new stack segment not present
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        NewSSP := IA32_PL0_SSP (* the new code-segment DPL must be 0 *)
        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)); FI;
        (* 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)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    FI;
    IF instruction pointer from IDT gate is not within new code-segment limits
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    tempEFLAGS := EFLAGS;
    VM := 0;
    TF := 0;
    RF := 0;
    NT := 0;
    IF service through interrupt gate
        THEN IF = 0; FI;
    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);
    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 := Gate(CS); (* Segment descriptor information also loaded *)
    CS(RPL) := 0;
    CPL := 0;
    IF IDT gate is 32-bit
        THEN
            EIP := Gate(instruction pointer);
        ELSE (* IDT gate is 16-bit *)
            EIP := Gate(instruction pointer) AND 0000FFFFH;
    FI;
    IF ShadowStackEnabled(0)
        oldSSP := SSP
        SSP := NewSSP
        IF SSP & 0x07 != 0
            THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
        IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)

```

```

        #GP(0); FI;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;
expected_token_value = SSP (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
    THEN #GP(0); FI;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
    IA32_S_CET.SUPPRESS = 0
FI;
(* Start execution of new routine in Protected Mode *)
END;

```

INTRA-PRIVILEGE-LEVEL-INTERRUPT:

```

NewSSP = SSP;
CHECK_SS_TOKEN = 0
(* 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);
            ELSE NewRSP := RSP;
        FI;
    IF IDT-descriptor IST ≠ 0
        IF ShadowStackEnabled(CPL)
            THEN
                NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT gate IST << 3)
                NewSSP := 8 bytes loaded from NewSSPAddress
                CHECK_SS_TOKEN = 1
            FI;
        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 & FFFFFFFF0H;
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 ShadowStackEnabled(CPL)
    IF CHECK_SS_TOKEN == 1
    THEN
        IF NewSSP & 0x07 != 0
        THEN #GP(0); FI;
        (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
        IF (NewSSP & ~0x1F) != ((NewSSP - 24) & ~0x1F)
        THEN #GP(0); FI;

        IF ((IA32_EFER.LMA and CS.L) = 0 AND NewSSP[63:32] != 0)
        THEN #GP(0); FI;
        expected_token_value = NewSSP (* busy bit - bit position 0 - must be clear *)
        new_token_value = NewSSP | BUSY_BIT (* Set the busy bit *)
        IF shadow_stack_lock_cmpxchg8b(NewSSP, new_token_value, expected_token_value) != expected_token_value
        THEN #GP(0); FI;
    FI;
    (* Align to next 8 byte boundary *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (NewSSP - 4)
    SSP = newSSP & 0xFFFFFFFFFFFF8H;
    (* push cs:lip:ssp on shadow stack *)

```

```

    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
FI;
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 vector selects a descriptor outside the IDT limits.

If an IDT descriptor is not an interrupt, trap, or task gate.

If an interrupt is generated by the INT *n*, INT3, or INTO instruction and the DPL of an interrupt, trap, or task gate 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.

If SSP in IA32_PLi_SSP (where *i* is the new CPL) is not 8 byte aligned.

If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.

If “supervisor Shadow Stack” token on new shadow stack is marked busy.

If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.

If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where *i* is the new CPL).

#SS(error_code)	<p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p> <p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>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.</p>
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code)	<p>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.</p> <p>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.</p> <p>If the stack segment selector in the TSS is NULL.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#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	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the interrupt vector number is outside the IDT limits.</p>
#SS	<p>If stack limit violation on push.</p> <p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.</p>
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	<p>(For INT n, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate is not equal to 3.</p> <p>If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt, trap, or task gate is NULL.</p> <p>If a interrupt gate, trap gate, task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt, trap, or task gate.</p> <p>If an interrupt is generated by INT n, INT3, or INTO and the DPL of an interrupt, trap, or task gate is less than the CPL.</p> <p>If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p>
#SS(error_code)	<p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.</p>
#NP(error_code)	If code segment, interrupt gate, trap gate, task gate, or TSS is not present.

#TS(error_code)	<p>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.</p> <p>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.</p> <p>If the stack segment selector in the TSS is NULL.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#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)	<p>If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical.</p> <p>If the segment selector in the 64-bit interrupt or trap gate is NULL.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If the vector points to a gate which is in non-canonical space.</p> <p>If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate.</p> <p>If the descriptor pointed to by the gate selector is outside the descriptor table limit.</p> <p>If the descriptor pointed to by the gate selector is in non-canonical space.</p> <p>If the descriptor pointed to by the gate selector is not a code segment.</p> <p>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.</p> <p>If the descriptor pointed to by the gate selector has DPL > CPL.</p> <p>If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.</p> <p>If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.</p> <p>If "supervisor shadow stack" token on new shadow stack is marked busy.</p> <p>If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4GB.</p> <p>If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).</p>
#SS(error_code)	<p>If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.</p> <p>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).</p>
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	<p>If an attempt to load RSP from the TSS causes an access to non-canonical space.</p> <p>If the RSP from the TSS is outside descriptor table limits.</p>
#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 ¹	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 08	INVD	Z0	Valid	Valid	Flush internal caches; initiate flushing of external caches.

NOTES:

1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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.
	If the processor reserved memory protections are activated.
#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 Entries

Opcode ¹	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01/7	INVLPG m	M	Valid	Valid	Invalidate TLB entries for page containing m.

NOTES:

1. 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)	N/A	N/A	N/A

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.¹

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 TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction invalidates TLB entries associated with the current PCID and may or may not do so for TLB entries associated with other PCIDs. (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” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B and 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.

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

Invalidate(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.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A), the instruction invalidates all of them.

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)	N/A	N/A

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*.) Invalidation is 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.¹ 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 3-25. 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.

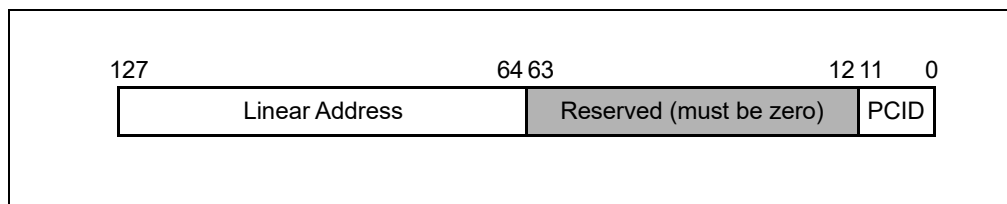


Figure 3-25. INVPCID Descriptor

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A), the instruction invalidates all of them.

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 IA-32e 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)	<p>If the current privilege level is not 0.</p> <p>If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the source operand is located in an execute-only code segment.</p> <p>If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.</p> <p>If bits 63:12 of INVPCID_DESC are not all zero.</p> <p>If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.</p> <p>If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	<p>If the memory operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	<p>If if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.</p> <p>If the LOCK prefix is used.</p>

Real-Address Mode Exceptions

#GP	<p>If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.</p> <p>If bits 63:12 of INVPCID_DESC are not all zero.</p> <p>If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.</p> <p>If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.</p>
#UD	<p>If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.</p> <p>If the LOCK prefix is used.</p>

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)	<p>If the current privilege level is not 0.</p> <p>If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.</p> <p>If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.</p> <p>If bits 63:12 of INVPCID_DESC are not all zero.</p> <p>If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.</p> <p>If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.</p>
#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	<p>If the LOCK prefix is used.</p> <p>If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.</p>

IRET/IRETD/IRETQ—Interrupt Return

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CF	IRET	ZO	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	ZO	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	ZO	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

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 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

IRET and IRETD are mnemonics for the same opcode. The IRETD 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.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, “Handling Multiple NMIs” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A), execution of the IRET instruction unblocks NMIs.

This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

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.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 17, "Control-Flow Enforcement Technology (CET)" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for CET details.

Instruction ordering. IRET is a serializing instruction. See Section 9.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 26 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;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE:
    IF OperandSize = 32
        THEN
            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 *)
            EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS := Pop(); (* 16-bit pop *)
            EFLAGS[15:0] := Pop();
        FI;
    END;

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
                EIP := Pop();
                CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS := Pop();
                (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS := Pop(); (* 16-bit pop *)
                EFLAGS[15:0] := Pop(); (* IOPL in EFLAGS not modified by pop *)
            FI;
        FI;
    FI;

```

```

        IF EIP not within CS limit
            THEN #GP(0); FI;
        FI;
    ELSE
        #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

```

PROTECTED-MODE:

```

    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS := Pop();
        ELSE (* OperandSize = 16 *)
            EIP := Pop(); (* 16-bit pop; clear upper bits *)
            CS := Pop(); (* 16-bit pop *)
            tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
        FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

```

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)

```

    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 CS limit
        THEN #GP(0); FI;
    FI;
END;

```

RETURN-TO-VIRTUAL-8086-MODE:

```

    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    (* If shadow stack or indirect branch tracking at CPL3 then #GP(0) *)
    IF CR4.CET AND (IA32_U_CET.ENDBR_EN OR IA32_U_CET.SHSTK_EN)
        THEN #GP(0); FI;
    shadowStackEnabled = ShadowStackEnabled(CPL)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS := tempEFLAGS;
    ESP := Pop();
    SS := Pop(); (* Pop 2 words; throw away high-order word *)
    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 *)
    IF shadowStackEnabled
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;

```

```

CPL := 3;
(* Resume execution in Virtual-8086 mode *)
tempOldSSP = SSP;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF shadowStackEnabled
    expected_token_value = tempOldSSP | BUSY_BIT    (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                    (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
            tempESP := Pop();
            tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE IF OperandSize = 16
            THEN
                tempESP := Pop(); (* 16-bit pop; clear upper bits *)
                tempSS := Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                tempRSP := Pop();
                tempSS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
            FI;
        IF new mode ≠ 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
                ELSE (* new mode = 64-bit mode *)
                    IF RIP is non-canonical
                        THEN #GP(0); FI;
                FI;
            EFLAGS(CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
            IF CPL ≤ IOPL
                THEN EFLAGS(IF) := tempEFLAGS; FI;
            IF CPL = 0
                THEN
                    EFLAGS(IOPL) := tempEFLAGS;
                    IF OperandSize = 32 or OperandSize = 64
                        THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
            FI;
            IF ShadowStackEnabled(CPL)
                (* check if 8 byte aligned *)
                IF SSP AND 0x7 != 0

```

```

    THEN #CP(FAR-RET/IRET); FI;
IF CS(RPL) != 3
    THEN
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        (* Do 64 bit compare to detect bits beyond 15 being set *)
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
        IF CSBASE + RIP != tempSsEIP
            THEN #CP(FAR-RET/IRET); FI;
        (* check if 4 byte aligned *)
        IF tempSSP AND 0x3 != 0
            THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;
tempOldCPL = CPL;
CPL := CS(RPL);
IF OperandSize = 64
    THEN
        RSP := tempRSP;
        SS := tempSS;
    ELSE
        ESP := tempESP;
        SS := tempSS;
    FI;
IF new mode != 64-Bit Mode
    THEN
        IF EIP is not within CS limit
            THEN #GP(0); FI;
    ELSE (* new mode = 64-bit mode *)
        IF RIP is non-canonical
            THEN #GP(0); FI;
    FI;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; FI;
IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
   ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
    THEN #GP(0); FI;
SSP := tempSSP;
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;

```

```

FOR each SegReg in (ES, FS, GS, and DS)
    DO
        tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
        IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
            THEN (* Segment register invalid *)
                SegmentSelector := 0; (*Segment selector becomes null*)
        FI;
    OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
    IF new mode ≠ 64-Bit Mode
        THEN
            IF EIP is not within CS limit
                THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
                IF RIP is non-canonical
                    THEN #GP(0); FI;
            FI;
            EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
            IF CPL ≤ IOPL
                THEN EFLAGS(IF) := tempEFLAGS; FI;
            IF CPL = 0
                THEN
                    EFLAGS(IOPL) := tempEFLAGS;
                    IF OperandSize = 32 or OperandSize = 64
                        THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
            FI;
            IF ShadowStackEnabled(CPL)
                IF SSP AND 0x7 ≠ 0 (* check if aligned to 8 bytes *)
                    THEN #CP(FAR-RET/IRET); FI;
                tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                tempSSP = shadow_stack_load 8 bytes from SSP;
                SSP = SSP + 24;
                tempCS = CS; (* zero padded to 64 bit *)
                IF tempCS ≠ tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
                    THEN #CP(FAR-RET/IRET); FI;
                IF CSBASE + RIP ≠ tempSsLIP (* 64 bit compare; CSBASE+RIP zero padded to 64 bit for 32 bit LIP *)
                    THEN #CP(FAR-RET/IRET); FI;
                IF tempSSP AND 0x3 ≠ 0 (* check if aligned to 4 bytes *)
                    THEN #CP(FAR-RET/IRET); FI;
                IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] ≠ 0) OR
                    ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
                    THEN #GP(0); FI;
            FI;
            IF ShadowStackEnabled(CPL)
                IF IA32_EFER.LMA = 1
                    (* In IA-32e-mode the IRET may be switching stacks if the interrupt/exception was delivered
                    through an IDT with a non-zero IST *)
                    (* In IA-32e mode for same CPL IRET there is always a stack switch. The below check verifies if the
                    stack switch was to self stack and if so, do not try to free the token on this shadow stack. If the

```

```

tempSSP was not to same stack then there was a stack switch so do attempt to free the token *)
  IF tempSSP != SSP
    THEN
      expected_token_value = SSP | BUSY_BIT      (* busy bit - bit position 0 - must be set *)
      new_token_value = SSP                      (* clear the busy bit *)
      shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)
    FI;
  FI;
  SSP := tempSSP
FI;
END;

IA-32e-MODE:
  IF NT = 1
    THEN #GP(0);
  ELSE IF OperandSize = 32
    THEN
      EIP := Pop();
      CS := Pop();
      tempEFLAGS := Pop();
    ELSE IF OperandSize = 16
      THEN
        EIP := Pop(); (* 16-bit pop; clear upper bits *)
        CS := Pop(); (* 16-bit pop *)
        tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
      FI;
    ELSE (* OperandSize = 64 *)
      THEN
        RIP := Pop();
        CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        tempRFLAGS := Pop();
      FI;
    IF CS.RPL > CPL
      THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE
      IF instruction began in 64-Bit Mode
        THEN
          IF OperandSize = 32
            THEN
              ESP := Pop();
              SS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE IF OperandSize = 16
              THEN
                ESP := Pop(); (* 16-bit pop; clear upper bits *)
                SS := Pop(); (* 16-bit pop *)
              ELSE (* OperandSize = 64 *)
                RSP := Pop();
                SS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
              FI;
            FI;
          GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
        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. If the return stack segment is not present.
#NP (selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP (Far-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned. If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB. If return instruction pointer from stack and shadow stack do not match.

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.
 If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
- #SS(0) If an attempt to pop a value off the stack violates the SS limit.
 If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
 If the return stack segment is not present.
- #NP (selector) If the return code segment is not present.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
- #UD If the LOCK prefix is used.
- #CP (Far-RET/IRET) If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
 If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
 If return instruction pointer from stack and shadow stack do not match.

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=0F).
7D cb	JGE rel8	D	Valid	Valid	Jump short if greater or equal (SF=0F).
7C cb	JL rel8	D	Valid	Valid	Jump short if less (SF≠ 0F).
7E cb	JLE rel8	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ 0F).
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≠ 0F).
7C cb	JNGE rel8	D	Valid	Valid	Jump short if not greater or equal (SF≠ 0F).
7D cb	JNL rel8	D	Valid	Valid	Jump short if not less (SF=0F).
7F cb	JNLE rel8	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=0F).
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	JA 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.
0F 83 cd	JAe rel32	D	Valid	Valid	Jump near if above or equal (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
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=0F). Not supported in 64-bit mode.
0F 8F cd	JG rel32	D	Valid	Valid	Jump near if greater (ZF=0 and SF=0F).
0F 8D cw	JGE rel16	D	N.S.	Valid	Jump near if greater or equal (SF=0F). Not supported in 64-bit mode.
0F 8D cd	JGE rel32	D	Valid	Valid	Jump near if greater or equal (SF=0F).
0F 8C cw	JL rel16	D	N.S.	Valid	Jump near if less (SF≠ 0F). Not supported in 64-bit mode.
0F 8C cd	JL rel32	D	Valid	Valid	Jump near if less (SF≠ 0F).
0F 8E cw	JLE rel16	D	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠ 0F). Not supported in 64-bit mode.
0F 8E cd	JLE rel32	D	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠ 0F).
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).
0F 85 cw	JNE rel16	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
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≠ 0F). Not supported in 64-bit mode.
0F 8E cd	JNG rel32	D	Valid	Valid	Jump near if not greater (ZF=1 or SF≠ 0F).
0F 8C cw	JNGE rel16	D	N.S.	Valid	Jump near if not greater or equal (SF≠ 0F). Not supported in 64-bit mode.
0F 8C cd	JNGE rel32	D	Valid	Valid	Jump near if not greater or equal (SF≠ 0F).
0F 8D cw	JNL rel16	D	N.S.	Valid	Jump near if not less (SF=0F). Not supported in 64-bit mode.
0F 8D cd	JNL rel32	D	Valid	Valid	Jump near if not less (SF=0F).
0F 8F cw	JNLE rel16	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=0F). 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=0F).
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.
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	N/A	N/A	N/A

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 JRCXZ, 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 cache-ability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is $RIP = RIP + 8\text{-bit offset sign extended to 64 bits}$. JMP Near is $RIP = RIP + 32\text{-bit offset sign extended to 64 bits}$.

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;

```

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 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	S	Inv.	Valid	Jump far, absolute, address given in operand.
EA cp	JMP ptr16:32	S	Inv.	Valid	Jump far, absolute, address given in operand.
FF /5	JMP m16:16	M	Valid	Valid	Jump far, absolute indirect, address given in m16:16.
FF /5	JMP m16:32	M	Valid	Valid	Jump far, absolute indirect, address given in m16:32.
REX.W FF /5	JMP m16:64	M	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
S	Segment + Absolute Address	N/A	N/A	N/A
D	Offset	N/A	N/A	N/A
M	ModRM:r/m (r)	N/A	N/A	N/A

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 8, 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 8 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 at 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.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 17, "Control-Flow Enforcement Technology (CET)" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for CET details.

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.

Instruction ordering. Instructions following a far jump may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far jump have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Instructions sequentially following a near indirect JMP instruction (i.e., those not at the target) may be executed speculatively. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an INT3 or LFENCE instruction opcode can be placed after the near indirect JMP in order to block speculative execution.

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;
IF (JMP near indirect, absolute indirect)
    IF EndbranchEnabledAndNotSuppressed(CPL)
        IF CPL = 3
            THEN
                IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                ELSE
                    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                        FI;
                    FI;
                FI;
            FI;
        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;
        FI;
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 (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); 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;
    IF ShadowStackEnabled(CPL)
        IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
            (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
            IF (SSP & 0xFFFFFFFF00000000 != 0)
                THEN #GP(0); FI;
        FI;
    FI;
    CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) := CPL
    EIP := tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;
NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) OR (DPL ≠ CPL)
        THEN #GP(code 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;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
        FI;
    FI;
FI;
CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
IF EndbranchEnabled(CPL)
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
FI;
END;

```

CALL-GATE:

```

IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); FI;
IF call gate not present
    THEN #NP(call gate selector); FI;
IF call gate code-segment selector is NULL
    THEN #GP(0); FI;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); FI;
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); FI;
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); FI;
IF code segment is not present
    THEN #NP(code-segment selector); FI;
tempEIP := DEST(Offset);
IF GateSize = 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
CS := DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;

```

```

IF EndbranchEnabled(CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;
END;
TASK-GATE:
  IF task gate DPL < CPL
  or task gate DPL < task gate segment-selector RPL
    THEN #GP(task gate selector); FI;
  IF task gate not present
    THEN #NP(gate selector); FI;
  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 descriptor is not a TSS segment
  or TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); FI;
  IF TSS not present
    THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
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); FI;
  IF TSS is not present
    THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
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. If target mode is compatibility mode and SSP is not in low 4GB.
--------	---

#GP(selector)	<p>If the segment selector index is outside descriptor table limits.</p> <p>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.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL</p> <p>(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>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.</p> <p>If the segment descriptor for selector in a call gate does not indicate it is a code segment.</p> <p>If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP (selector)	<p>If the code segment being accessed is not present.</p> <p>If call gate, task gate, or TSS not present.</p>
#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	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p>
#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)	<p>If the target operand is beyond the code segment limits.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p>
#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)	<p>If a memory address is non-canonical.</p> <p>If target offset in destination operand is non-canonical.</p> <p>If target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the 64-bit gate is NULL.</p> <p>If transitioning to compatibility mode and the SSP is beyond 4GB.</p>
--------	--

#GP(selector)	<p>If the code segment or 64-bit call gate is outside descriptor table limits.</p> <p>If the code segment or 64-bit call gate overlaps non-canonical space.</p> <p>If the segment descriptor from a 64-bit call gate is in non-canonical space.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.</p> <p>If the upper type field of a 64-bit call gate is not 0x0.</p> <p>If the segment selector from a 64-bit call gate is beyond the descriptor table limits.</p> <p>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.</p> <p>If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.</p> <p>If the code segment is non-conforming and $CPL \neq DPL$.</p> <p>If the code segment is confirming and $CPL < DPL$.</p>
#NP(selector)	If a code segment or 64-bit call gate is not present.
#UD	<p>(64-bit mode only) If a far jump is direct to an absolute address in memory.</p> <p>If the LOCK prefix is used.</p>
#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.

KADDW/KADDB/KADDQ/KADD—ADD Two Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 4A /r KADDW k1, k2, k3	RVR	V/V	AVX512DQ	Add 16 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3	RVR	V/V	AVX512DQ	Add 8 bits masks in k2 and k3 and place result in k1.
VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3	RVR	V/V	AVX512BW	Add 64 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 4A /r KADD k1, k2, k3	RVR	V/V	AVX512BW	Add 32 bits masks in k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KADDW

DEST[15:0] := SRC1[15:0] + SRC2[15:0]
DEST[MAX_KL-1:16] := 0

KADDB

DEST[7:0] := SRC1[7:0] + SRC2[7:0]
DEST[MAX_KL-1:8] := 0

KADDQ

DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[MAX_KL-1:64] := 0

KADD

DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KADDW __mmask16 _kadd_mask16 (__mmask16 a, __mmask16 b);
KADDB __mmask8 _kadd_mask8 (__mmask8 a, __mmask8 b);
KADDQ __mmask64 _kadd_mask64 (__mmask64 a, __mmask64 b);
KADD __mmask32 _kadd_mask32 (__mmask32 a, __mmask32 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 41 /r KANDB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 41 /r KANDQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 41 /r KANDD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDW

DEST[15:0] := SRC1[15:0] BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] := 0

KANDB

DEST[7:0] := SRC1[7:0] BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] := 0

KANDQ

DEST[63:0] := SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] := 0

KANDD

DEST[31:0] := SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KANDW __mmask16 _mm512_kand(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 42 /r KANDNB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.
VEX.L1.0F.W1 42 /r KANDNQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 42 /r KANDND k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation**KANDNW**

DEST[15:0] := (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]
 DEST[MAX_KL-1:16] := 0

KANDNB

DEST[7:0] := (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]
 DEST[MAX_KL-1:8] := 0

KANDNQ

DEST[63:0] := (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]
 DEST[MAX_KL-1:64] := 0

KANDND

DEST[31:0] := (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]
 DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KANDNW __mmask16 __mm512_kandn(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KMOVW/KMOVB/KMOVQ/KMOVD—Move From and to Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16	RM	V/V	AVX512F	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8	RM	V/V	AVX512DQ	Move 8 bits mask from k2/m8 and store the result in k1.
VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64	RM	V/V	AVX512BW	Move 64 bits mask from k2/m64 and store the result in k1.
VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32	RM	V/V	AVX512BW	Move 32 bits mask from k2/m32 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.66.0F.W0 91 /r KMOVB m8, k1	MR	V/V	AVX512DQ	Move 8 bits mask from k1 and store the result in m8.
VEX.L0.0F.W1 91 /r KMOVQ m64, k1	MR	V/V	AVX512BW	Move 64 bits mask from k1 and store the result in m64.
VEX.L0.66.0F.W1 91 /r KMOVD m32, k1	MR	V/V	AVX512BW	Move 32 bits mask from k1 and store the result in m32.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F	Move 16 bits mask from r32 to k1.
VEX.L0.66.0F.W0 92 /r KMOVB k1, r32	RR	V/V	AVX512DQ	Move 8 bits mask from r32 to k1.
VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64	RR	V/I	AVX512BW	Move 64 bits mask from r64 to k1.
VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32	RR	V/V	AVX512BW	Move 32 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F	Move 16 bits mask from k1 to r32.
VEX.L0.66.0F.W0 93 /r KMOVB r32, k1	RR	V/V	AVX512DQ	Move 8 bits mask from k1 to r32.
VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1	RR	V/I	AVX512BW	Move 64 bits mask from k1 to r64.
VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1	RR	V/V	AVX512BW	Move 32 bits mask from k1 to r32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RM	ModRM:reg (w)	ModRM:r/m (r)
MR	ModRM:r/m (w, ModRM:[7:6] must not be 11b)	ModRM:reg (r)
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that VEX.W can only be used to modify the size of the GPR operand in 64b mode.

Operation

KMOVW

IF *destination is a memory location*

DEST[15:0] := SRC[15:0]

IF *destination is a mask register or a GPR *

DEST := ZeroExtension(SRC[15:0])

KMOVB

IF *destination is a memory location*

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

IF *destination is a mask register or a GPR *

DEST := ZeroExtension(SRC[7:0])

KMOVQ

IF *destination is a memory location or a GPR*

DEST[63:0] := SRC[63:0]

IF *destination is a mask register*

DEST := ZeroExtension(SRC[63:0])

KMOVD

IF *destination is a memory location*

DEST[31:0] := SRC[31:0]

IF *destination is a mask register or a GPR *

DEST := ZeroExtension(SRC[31:0])

Intel C/C++ Compiler Intrinsic Equivalent

KMOVW __mmask16 _mm512_kmov(__mmask16 a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Instructions with RR operand encoding, see Table 2-63, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

Instructions with RM or MR operand encoding, see Table 2-64, "TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)."

KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 32 bits mask k2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

Operation**KNOTW**

DEST[15:0] := BITWISE NOT SRC[15:0]

DEST[MAX_KL-1:16] := 0

KNOTB

DEST[7:0] := BITWISE NOT SRC[7:0]

DEST[MAX_KL-1:8] := 0

KNOTQ

DEST[63:0] := BITWISE NOT SRC[63:0]

DEST[MAX_KL-1:64] := 0

KNOTD

DEST[31:0] := BITWISE NOT SRC[31:0]

DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KNOTW __mmask16 _mm512_knot(__mmask16 a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise OR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise OR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 45 /r KORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation**KORW**

DEST[15:0] := SRC1[15:0] BITWISE OR SRC2[15:0]
DEST[MAX_KL-1:16] := 0

KORB

DEST[7:0] := SRC1[7:0] BITWISE OR SRC2[7:0]
DEST[MAX_KL-1:8] := 0

KORQ

DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAX_KL-1:64] := 0

KORD

DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KORW __mmask16 __mm512_kor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks and Set Flags

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512F	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2	RR	V/V	AVX512DQ	Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.0F.W1 98 /r KORTESTQ k1, k2	RR	V/V	AVX512BW	Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2	RR	V/V	AVX512BW	Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

Operation**KORTESTW**

TMP[15:0] := DEST[15:0] BITWISE OR SRC[15:0]

IF(TMP[15:0]=0)

THEN ZF := 1

ELSE ZF := 0

FI;

IF(TMP[15:0]=FFFFh)

THEN CF := 1

ELSE CF := 0

FI;

KORTESTB

TMP[7:0] := DEST[7:0] BITWISE OR SRC[7:0]

IF(TMP[7:0]=0)

THEN ZF := 1

ELSE ZF := 0

FI;

IF(TMP[7:0]==FFh)

THEN CF := 1

ELSE CF := 0

FI;

KORTESTQ

TMP[63:0] := DEST[63:0] BITWISE OR SRC[63:0]

IF(TMP[63:0]=0)

THEN ZF := 1

ELSE ZF := 0

FI;

IF(TMP[63:0]==FFFFFFFF_FFFFFFFFh)

THEN CF := 1

ELSE CF := 0

FI;

KORTESTD

TMP[31:0] := DEST[31:0] BITWISE OR SRC[31:0]

IF(TMP[31:0]=0)

THEN ZF := 1

ELSE ZF := 0

FI;

IF(TMP[31:0]=FFFFFFFFh)

THEN CF := 1

ELSE CF := 0

FI;

Intel C/C++ Compiler Intrinsic Equivalent

KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);

Flags Affected

The ZF flag is set if the result of OR-ing both sources is all 0s.

The CF flag is set if the result of OR-ing both sources is all 1s.

The OF, SF, AF, and PF flags are set to 0.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8	RRI	V/V	AVX512F	Shift left 16 bits in k2 by immediate and write result in k1.
VEEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift left 8 bits in k2 by immediate and write result in k1.
VEEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 64 bits in k2 by immediate and write result in k1.
VEEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 32 bits in k2 by immediate and write result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTLW

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 15
    THEN DEST[15:0] := SRC1[15:0] << COUNT;
FI;
```

KSHIFTLB

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 7
    THEN DEST[7:0] := SRC1[7:0] << COUNT;
FI;
```

KSHIFTLQ

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 63
    THEN DEST[63:0] := SRC1[63:0] << COUNT;
FI;
```

KSHIFTLD

COUNT := imm8[7:0]

DEST[MAX_KL-1:0] := 0

IF COUNT <=31

 THEN DEST[31:0] := SRC1[31:0] << COUNT;
FI;

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTLW when needed.

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 30 /r KSHIFTRW k1, k2, imm8	RRI	V/V	AVX512F	Shift right 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift right 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 32 bits in k2 by immediate and write result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation**KSHIFTRW**

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 15
    THEN DEST[15:0] := SRC1[15:0] >> COUNT;
FI;
```

KSHIFTRB

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 7
    THEN DEST[7:0] := SRC1[7:0] >> COUNT;
FI;
```

KSHIFTRQ

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 63
    THEN DEST[63:0] := SRC1[63:0] >> COUNT;
FI;
```

KSHIFTRD

```
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <= 31
    THEN    DEST[31:0] := SRC1[31:0] >> COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTRW when needed.

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 99 /r KTESTW k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.
VEX.L0.66.0F.W0 99 /r KTESTB k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.
VEX.L0.0F.W1 99 /r KTESTQ k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.
VEX.L0.66.0F.W1 99 /r KTESTD k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

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

Operation**KTESTW**

TEMP[15:0] := SRC2[15:0] AND SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN ZF := 1;

ELSE ZF := 0;

FI;

TEMP[15:0] := SRC2[15:0] AND NOT SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN CF := 1;

ELSE CF := 0;

FI;

AF := OF := PF := SF := 0;

KTESTB

TEMP[7:0] := SRC2[7:0] AND SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN ZF := 1;

ELSE ZF := 0;

FI;

TEMP[7:0] := SRC2[7:0] AND NOT SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN CF := 1;

ELSE CF := 0;

FI;

AF := OF := PF := SF := 0;

KTESTQ

```

TEMP[63:0] := SRC2[63:0] AND SRC1[63:0]
IF (TEMP[63:0] == 0)
    THEN ZF := 1;
    ELSE ZF := 0;
FI;
TEMP[63:0] := SRC2[63:0] AND NOT SRC1[63:0]
IF (TEMP[63:0] == 0)
    THEN CF := 1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

```

KTESTD

```

TEMP[31:0] := SRC2[31:0] AND SRC1[31:0]
IF (TEMP[31:0] == 0)
    THEN ZF := 1;
    ELSE ZF := 0;
FI;
TEMP[31:0] := SRC2[31:0] AND NOT SRC1[31:0]
IF (TEMP[31:0] == 0)
    THEN CF := 1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

```

Intel C/C++ Compiler Intrinsic Equivalent**SIMD Floating-Point Exceptions**

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F	Unpack 8-bit masks in k2 and k3 and write word result in k1.
VEX.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3	RVR	V/V	AVX512BW	Unpack 16-bit masks in k2 and k3 and write doubleword result in k1.
VEX.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3	RVR	V/V	AVX512BW	Unpack 32-bit masks in k2 and k3 and write quadword result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

Operation

KUNPCKBW

```
DEST[7:0] := SRC2[7:0]
DEST[15:8] := SRC1[7:0]
DEST[MAX_KL-1:16] := 0
```

KUNPCKWD

```
DEST[15:0] := SRC2[15:0]
DEST[31:16] := SRC1[15:0]
DEST[MAX_KL-1:32] := 0
```

KUNPCKDQ

```
DEST[31:0] := SRC2[31:0]
DEST[63:32] := SRC1[31:0]
DEST[MAX_KL-1:64] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
KUNPCKBW __mmask16 __mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 __mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 __mm512_kunpackw(__mmask32 a, __mmask32 b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."

KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XNOR 16-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 46 /r KXNORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XNOR 8-bit masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 46 /r KXNORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 64-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 46 /r KXNORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 32-bit masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXNORW

DEST[15:0] := NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])
DEST[MAX_KL-1:16] := 0

KXNORB

DEST[7:0] := NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])
DEST[MAX_KL-1:8] := 0

KXNORQ

DEST[63:0] := NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])
DEST[MAX_KL-1:64] := 0

KXNORD

DEST[31:0] := NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])
DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KXNORW __mmask16 __mm512_kxnor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XOR 16-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XOR 8-bit masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 64-bit masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 32-bit masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXORW

DEST[15:0] := SRC1[15:0] BITWISE XOR SRC2[15:0]
DEST[MAX_KL-1:16] := 0

KXORB

DEST[7:0] := SRC1[7:0] BITWISE XOR SRC2[7:0]
DEST[MAX_KL-1:8] := 0

KXORQ

DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAX_KL-1:64] := 0

KXORD

DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent

KXORW __mmask16 _mm512_kxor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

LAHF—Load Status Flags Into AH Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9F	LAHF	Z0	Invalid ¹	Valid	Load: AH := EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

NOTES:

1. Valid in specific steppings; see Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 r16, r16/m16	RM	Valid	Valid	r16 := access rights referenced by r16/m16
OF 02 /r	LAR reg, r32/m16 ¹	RM	Valid	Valid	reg := access rights referenced by r32/m16

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)	N/A	N/A

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 3-53.
- 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 3-53. 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	Yes
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 0F F0 /r LDDQU <i>xmm1</i> , <i>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</i> , <i>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</i> , <i>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)	N/A	N/A

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[MAXVL-1:128] (Unmodified)

VLDDQU (VEX.128 Encoded Version)

DEST[127:0] := SRC[127:0]

DEST[MAXVL-1:128] := 0

VLDDQU (VEX.256 Encoded Version)

DEST[255:0] := SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU __m128i _mm_lddqu_si128 (__m128i * p);

VLDDQU __m256i _mm256_lddqu_si256 (__m256i * p);

Numeric Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”

Note treatment of #AC varies.

LDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP OF AE /2 LDMXCSR m32	M	V/V	SSE	Load MXCSR register from m32.
VEX.LZ.OF.WIG AE /2 VLDMXCSR m32	M	V/V	AVX	Load MXCSR register from m32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

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

MXCSR := m32;

C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

Numeric Exceptions

None.

Other Exceptions

See Table 2-22, “Type 5 Class Exception Conditions,” 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)	N/A	N/A

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(selector); FI;
    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); FI;
    ELSE IF Segment marked not present
        THEN #SS(selector); FI;
    FI;
    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); FI;
    ELSE IF Segment marked not present
        THEN #NP(selector); FI;
    FI;
    SegmentRegister := SegmentSelector(SRC) ;
    SegmentRegister := SegmentDescriptor([SRC]);
FI;
ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister := NULLSelector;
        SegmentRegister(DescriptorValidBit) := 0; FI; (* Hidden flag;
            not accessible by software *)
FI;
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); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS := SegmentSelector(SRC);
        SS := SegmentDescriptor([SRC]);
    ELSE IF DS, ES, 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); FI;
        ELSE IF Segment marked not present

```

```

        THEN #NP(selector); FI;
    FI;
    SegmentRegister := SegmentSelector(SRC) AND RPL;
    SegmentRegister := SegmentDescriptor([SRC]);
FI;
ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister := NULLSelector;
        SegmentRegister(DescriptorValidBit) := 0; FI; (* Hidden flag;
            not accessible by software *)
    FI;
    DEST := Offset(SRC);
Real-Address or Virtual-8086 Mode
    SegmentRegister := SegmentSelector(SRC); FI;
    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 NULL 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 NULL 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-NULL 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-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.

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.

LDTILECFG—Load Tile Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.OF38.W0 49 ! (11):000:bbb LDTILECFG m512	A	V/N.E.	AMX-TILE	Load tile configuration as specified in m512.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r)	N/A	N/A	N/A

Description

The LDTILECFG instruction takes an operand containing a pointer to a 64-byte memory location containing the description of the tiles to be supported. In order to configure the tiles, the AMX-TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

The memory area contains the palette and describes how many tiles are being used and defines each tile in terms of rows and column bytes. Requests must be compatible with the restrictions provided by CPUID; see Table 3-10 below.

Table 3-10. Memory Area Layout

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used.
1	start_row	start_row is used for storing the restart values for interrupted operations.
2-15	reserved, must be zero	
16-17	tile0.colsb	Tile 0 bytes per row.
18-19	tile1.colsb	Tile 1 bytes per row.
20-21	tile2.colsb	Tile 2 bytes per row.
...	(sequence continues)	
30-31	tile7.colsb	Tile 7 bytes per row.
32-47	reserved, must be zero	
48	tile0.rows	Tile 0 rows.
49	tile1.rows	Tile 1 rows.
50	tile2.rows	Tile 2 rows.
...	(sequence continues)	
55	tile7.rows	Tile 7 rows.
56-63	reserved, must be zero	

If a tile row and column pair is not used to specify tile parameters, they must have the value zero. All enabled tiles (based on the palette) must be configured. Specifying tile parameters for more tiles than the implementation limit or the palette limit results in a #GP fault.

If the palette_id is zero, that signifies the INIT state for both TILECFG and TILEDATA. Tiles are zeroed in the INIT state. The only legal non-INIT value for palette_id is 1.

Any attempt to execute the LDTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.

Operation**LDTILECFG mem**

```

error := False
buf := read_memory(mem, 64)
temp_tilecfg.palette_id := buf.byte[0]
if temp_tilecfg.palette_id > max_palette:
    error := True
if not xcr0_supports_palette(temp_tilecfg.palette_id):
    error := True
if temp_tilecfg.palette_id != 0:
    temp_tilecfg.start_row := buf.byte[1]
    if buf.byte[2..15] is nonzero:
        error := True
    p := 16
    # configure columns
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        temp_tilecfg.t[n].colsb := buf.word[p/2]
        p := p + 2
        if temp_tilecfg.t[n].colsb > palette_table[temp_tilecfg.palette_id].bytes_per_row:
            error := True
    if nonzero(buf[p...47]):
        error := True

    # configure rows
    p := 48
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        temp_tilecfg.t[n].rows := buf.byte[p]
        if temp_tilecfg.t[n].rows > palette_table[temp_tilecfg.palette_id].max_rows:
            error := True
        p := p + 1

    if nonzero(buf[p...63]):
        error := True

    # validate each tile's row & col configs are reasonable and enable the valid tiles
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        if temp_tilecfg.t[n].rows != 0 and temp_tilecfg.t[n].colsb != 0:
            temp_tilecfg.t[n].valid := 1
        elif temp_tilecfg.t[n].rows == 0 and temp_tilecfg.t[n].colsb == 0:
            temp_tilecfg.t[n].valid := 0
        else:
            error := True // one of rows or colsb was 0 but not both.

if error:
    #GP
elif temp_tilecfg.palette_id == 0:
    TILES_CONFIGURED := 0 // init state
    tilecfg := 0 // equivalent to 64B of zeros
    zero_all_tile_data()
else:
    tilecfg := temp_tilecfg
    zero_all_tile_data()
    TILES_CONFIGURED := 1

```

Intel C/C++ Compiler Intrinsic Equivalent

LDTILECFG void _tile_loadconfig(const void *);

Flags Affected

None.

Exceptions

AMX-E1; see Section 2.10, “Intel® AMX Instruction Exception Classes” for details.

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 m in register r16.
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for m in register r32.
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for m in register r64.

Instruction Operand Encoding

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

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 processors 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 3-54. 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 3-55.

Table 3-55. 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	Z0	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	Z0	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	Z0	Valid	N.E.	Set RSP to RBP, then pop RBP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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; FI;
    ELSE IF StackAddressSize = 16
        THEN SP := BP; FI;
FI;

IF OperandSize = 32
    THEN EBP := Pop();
    ELSE IF OperandSize = 64
        THEN RBP := Pop(); FI;
    ELSE IF OperandSize = 16
        THEN BP := Pop(); FI;
FI;

```

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	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE E8 LFENCE	Z0	V/V	SSE2	Serializes load operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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 (even speculatively) 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.

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

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 m16&32	M	N.E.	Valid	Load m into GDTR.
0F 01 /3	LIDT m16&32	M	N.E.	Valid	Load m into IDTR.
0F 01 /2	LGDT m16&64	M	Valid	N.E.	Load m into GDTR.
0F 01 /3	LIDT m16&64	M	Valid	N.E.	Load m into IDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

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 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 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 the LOCK prefix is used.
#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 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 r/m16 into LDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

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); FI;

IF segment selector is valid

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ LDT
THEN #GP(segment selector); FI;

IF segment descriptor is not present
THEN #NP(segment selector); FI;

LDTR(SegmentSelector) := SRC;

LDTR(SegmentDescriptor) := GDTSegmentDescriptor;

ELSE LDTR := INVALID

FI;

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 r/m16	M	Valid	Valid	Loads r/m16 in machine status word of CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

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 IA-32 and Intel 64 processors beginning with 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 26 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

CR0[0:3] := 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)	The LMSW instruction 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

#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.

LOADIWKEY—Load Internal Wrapping Key With Key Locker

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 DC 11:rrr:bbb LOADIWKEY xmm1, xmm2, <EAX>, <XMM0>	A	V/V	KL	Load internal wrapping key from xmm1, xmm2, and XMM0.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r)	ModRM:r/m (r)	Implicit EAX (r)	Implicit XMM0 (r)

Description

The `LOADIWKEY`¹ instruction writes the Key Locker internal wrapping key, which is called IWKey. This IWKey is used by the `ENCODEKEY*` instructions to wrap keys into handles. Conversely, the `AESENC/DEC*KL` instructions use IWKey to unwrap those keys from the handles and help verify the handle integrity. For security reasons, no instruction is designed to allow software to directly read the IWKey value.

IWKey includes two cryptographic keys as well as metadata. The two cryptographic keys are loaded from register sources so that `LOADIWKEY` can be executed without the keys ever being in memory.

The key input operands are:

- The 256-bit encryption key is loaded from the two explicit operands.
- The 128-bit integrity key is loaded from the implicit operand XMM0.

The implicit operand EAX specifies the KeySource and whether backing up the key is permitted:

- EAX[0] – When set, the wrapping key being initialized is not permitted to be backed up to platform-scoped storage.
- EAX[4:1] – This specifies the KeySource, which is the type of key. Currently only two encodings are supported. A KeySource of 0 indicates that the key input operands described above should be directly stored as the internal wrapping keys. `LOADIWKEY` with a KeySource of 1 will have random numbers from the on-chip random number generator XORed with the source registers (including XMM0) so that the software that executes the `LOADIWKEY` does not know the actual IWKey encryption and integrity keys. Software can choose to put additional random data into the source registers so that other sources of random data are combined with the hardware random number generator supplied value. Software should always check ZF after executing `LOADIWKEY` with KeySource of 1 as this operation may fail due to it being unable to get sufficient full-entropy data from the on-chip random number generator. Both KeySource of 0 and 1 specify that IWKey be used with the AES-GCM-SIV algorithm. CPUID.19H.ECX[1] enumerates support for KeySource of 1. All other KeySource encodings are reserved.
- EAX[31:5] – Reserved.

1. Further details on Key Locker and usage of this instruction can be found here:

<https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html>.

Operation

LOADIWKEY

```

IF CPL > 0                // LOADKWKEY only allowed at ring 0 (supervisor mode)
    THEN #GP (0); FI;
IF EAX[4:1] > 1           // Reserved KeySource encoding used
    THEN #GP (0); FI;
IF EAX[31:5] != 0         // Reserved bit in EAX is set
    THEN #GP (0); FI;
IF EAX[0] AND (CPUID.19H.ECX[0] == 0)      // NoBackup is not supported on this part
    THEN #GP (0); FI;
IF (EAX[4:1] == 1) AND (CPUID.19H.ECX[1] == 0) // KeySource of 1 is not supported on this part
    THEN #GP (0); FI;
IF (EAX[4:1] == 0)        // KeySource of 0
    THEN
        IwKey.Encryption Key[127:0] := SRC2[127:0];
        IwKey.Encryption Key[255:128] := SRC1[127:0];
        IwKey.IntegrityKey[127:0] := XMM0[127:0];
        IwKey.NoBackup = EAX [0];
        IwKey.KeySource = EAX [4:1];
        RFLAGS.ZF := 0;
    ELSE
        // KeySource of 1. See RDSEED definition for details of randomness
        IF HW_NRND_GEN.ready == 1           // Full-entropy random data from RDSEED hardware block was received
            THEN
                IwKey.Encryption Key[127:0] := SRC2[127:0] XOR HW_NRND_GEN.data[127:0];
                IwKey.Encryption Key[255:128] := SRC1[127:0] XOR HW_NRND_GEN.data[255:128];
                IwKey.IntegrityKey[127:0] := XMM0[127:0] XOR HW_NRND_GEN.data[383:256];
                IwKey.NoBackup = EAX [0];
                IwKey.KeySource = EAX [4:1];
                RFLAGS.ZF := 0;
            ELSE
                // Random data was not returned from RDSEED hardware block. IwKey was not loaded
                RFLAGS.ZF := 1;
        FI;
    FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

```

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to full-entropy random data not being received from RDSEED. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

```
LOADIWKEY    void _mm_loadikey(unsigned int ctl, __m128i intkey, __m128i enkey_lo, __m128i enkey_hi);
```

Exceptions (All Operating Modes)

#GP	<div>If $CPL > 0$. (Does not apply in real-address mode.) If $EAX[4:1] > 1$. If $EAX[31:5] \neq 0$. If $(EAX[0] == 1) \text{ AND } (CPUID.19H.ECX[0] == 0)$. If $(EAX[4:1] == 1) \text{ AND } (CPUID.19H.ECX[1] == 0)$.</div>
#UD	<div>If the LOCK prefix is used. If $CPUID.07H:ECX.KL [\text{bit } 23] = 0$. If $CR4.KL = 0$. If $CR0.EM = 1$. If $CR4.OSFXSR = 0$.</div>
#NM	<div>If $CR0.TS = 1$.</div>

LOCK—Assert LOCK# Signal Prefix

Opcode ¹	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F0	LOCK	Z0	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

NOTES:

1. See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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.

In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below for more details.

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 9 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, 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 m8	Z0	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 m16	Z0	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 m32	Z0	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 m64	Z0	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	Z0	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	Z0	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	Z0	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	Z0	Valid	N.E.	Load qword at address (R)SI into RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	N/A	N/A	N/A	N/A

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/REPZ—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;
    IF;
    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 \neq 0.
E1 cb	LOOPE rel8	D	Valid	Valid	Decrement count; jump short if count \neq 0 and ZF = 1.
E0 cb	LOOPNE rel8	D	Valid	Valid	Decrement count; jump short if count \neq 0 and ZF = 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	N/A	N/A	N/A

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  $\neq$  0)
                THEN BranchCond := 1;
                ELSE BranchCond := 0;
            FI;
        ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
            IF (ZF = 0) and (Count  $\neq$  0)
                THEN BranchCond := 1;
                ELSE BranchCond := 0;
            FI;
    
```

```

    FI;
ELSE (* Instruction = LOOP *)
    IF (Count ≠ 0)
        THEN BranchCond := 1;
        ELSE BranchCond := 0;
    FI;
FI;

IF BranchCond = 1
    THEN
        IF in 64-bit mode (* OperandSize = 64 *)
            THEN
                tempRIP := RIP + SignExtend(DEST);
                IF tempRIP is not canonical
                    THEN #GP(0);
                ELSE RIP := tempRIP;
                FI;
            ELSE
                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;
        ELSE
            Terminate loop and continue program execution at (R/E)IP;
    FI;

```

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 := segment limit, selector r16/m16.
OF 03 /r	LSL r32, r32/m16 ¹	RM	Valid	Valid	Load: r32 := segment limit, selector r32/m16.
REX.W + OF 03 /r	LSL r64, r32/m16 ¹	RM	Valid	Valid	Load: r64 := segment limit, selector r32/m16

NOTES:

1. 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)	N/A	N/A

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 3-56. Segment and Gate Descriptor 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 ¹	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 ¹	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS ¹	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

NOTES:

1. In this case, the descriptor comprises 16 bytes; bits 12:8 of the upper 4 bytes must be 0.

Operation

IF SRC(Offset) > descriptor table limit
THEN ZF := 0; FI;

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 (SegmentDescriptor(G) = 1)

THEN temp := (temp << 12) OR 00000FFFH;

ELSE IF OperandSize = 32

THEN DEST := temp; FI;

ELSE IF OperandSize = 64 (* REX.W used *)

THEN DEST := temp(* Zero-extended *); FI;

ELSE (* OperandSize = 16 *)

DEST := temp AND FFFFH;

FI;

FI;

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 r/m16	M	Valid	Valid	Load r/m16 into task register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

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); FI;

Read segment descriptor;

IF segment descriptor is not for an available TSS

THEN #GP(segment selector); FI;

IF segment descriptor is not present

THEN #NP(segment selector); FI;

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)	<p>If the current privilege level is not 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the source operand contains a NULL segment selector.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If the source selector points to a segment that is not a TSS or to one for a task that is already busy.</p> <p>If the selector points to LDT or is beyond the GDT limit.</p>
#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)	<p>If the current privilege level is not 0.</p> <p>If the memory address is in a non-canonical form.</p> <p>If the source operand contains a NULL segment selector.</p>
#GP(selector)	<p>If the source selector points to a segment that is not a TSS or to one for a task that is already busy.</p> <p>If the selector points to LDT or is beyond the GDT limit.</p> <p>If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.</p>
#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 0F 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 0F BD /r LZCNT r32, r/m32	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
F3 REX.W 0F 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)	N/A	N/A

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.
#UD	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)	For an illegal address in the SS segment.
#UD	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)	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.
#UD	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 LOCK prefix is used.

