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

Volume 2B: Instruction Set Reference, M-U

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers,* Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 253667-070US
May 2019

CHAPTER 4
INSTRUCTION SET REFERENCE, M-U

4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. The operation of the immediate control byte is common to these four string text processing instructions of SSE4.2. This section describes the common operations.

4.1.1 General Description

The operation of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM is defined by the combination of the respective opcode and the interpretation of an immediate control byte that is part of the instruction encoding.

The opcode controls the relationship of input bytes/words to each other (determines whether the inputs terminated strings or whether lengths are expressed explicitly) as well as the desired output (index or mask).

The Imm8 Control Byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

The arithmetic flags (ZF, CF, SF, OF, AF, PF) are set as a result of these instructions. However, the meanings of the flags have been overloaded from their typical meanings in order to provide additional information regarding the relationships of the two inputs.

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The Imm8 Control Byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the Imm8 Control Byte consists of bit fields that control the following attributes:

- **Source data format —** Byte/word data element granularity, signed or unsigned elements
- **Aggregation operation —** Encodes the mode of per-element comparison operation and the aggregation of per-element comparisons into an intermediate result
- **Polarity —** Specifies intermediate processing to be performed on the intermediate result
- **Output selection —** Specifies final operation to produce the output (depending on index or mask) from the intermediate result

4.1.2 Source Data Format

Table 4-1. Source Data Format

Imm8[1:0]	Meaning	Description
00b	Unsigned bytes	Both 128-bit sources are treated as packed, unsigned bytes.
01b	Unsigned words	Both 128-bit sources are treated as packed, unsigned words.
10b	Signed bytes	Both 128-bit sources are treated as packed, signed bytes.
11b	Signed words	Both 128-bit sources are treated as packed, signed words.

If the Imm8 Control Byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the Imm8 Control Byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

Imm8[3:2]	Mode	Comparison
00b	Equal any	The arithmetic comparison is “equal.”
01b	Ranges	Arithmetic comparison is “greater than or equal” between even indexed bytes/words of reg and each byte/word of reg/mem. Arithmetic comparison is “less than or equal” between odd indexed bytes/words of reg and each byte/word of reg/mem. (reg/mem[m] >= reg[n] for n = even, reg/mem[m] <= reg[n] for n = odd)
10b	Equal each	The arithmetic comparison is “equal.”
11b	Equal ordered	The arithmetic comparison is “equal.”

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by “BoolRes[*Reg/Mem element index, Reg element index*].” Comparisons evaluating to “True” are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by Imm8 Control Byte bit[3:2].

See Section 4.1.6 for a description of the overrideIfDataInvalid() function used in Table 4-3.

Table 4-3. Aggregation Operation

Mode	Pseudocode

Equal any
(find characters from a set)
	UpperBound = imm8[0] ? 7 : 15;
	IntRes1 = 0;
	For j = 0 to UpperBound, j++
	  For i = 0 to UpperBound, i++
	    IntRes1[j] OR= overrideIfDataInvalid(BoolRes[j,i])

Ranges
(find characters from ranges)
	UpperBound = imm8[0] ? 7 : 15;
	IntRes1 = 0;
	For j = 0 to UpperBound, j++
	  For i = 0 to UpperBound, i+=2
	    IntRes1[j] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))

Equal each (string compare)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For i = 0 to UpperBound, i++ IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])

Equal ordered
(substring search)
	UpperBound = imm8[0] ? 7 :15;
	IntRes1 = imm8[0] ? FFH : FFFFH
	For j = 0 to UpperBound, j++
	  For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++
	    IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])

4.1.4 Polarity

IntRes1 may then be further modified by performing a 1's complement, according to the value of the Imm8 Control Byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to "valid" reg/mem input elements are complemented (note that the definition of a valid input element is dependant on the specific opcode and is defined in each opcode's description). The result of the possible negation is referred to as IntRes2.

Table 4-4. Polarity

Imm8[5:4]	Operation	Description
00b	Positive Polarity (+)	IntRes2 = IntRes1
01b	Negative Polarity (-)	IntRes2 = -1 XOR IntRes1
10b	Masked (+)	IntRes2 = IntRes1
11b	Masked (-)	IntRes2[i] = IntRes1[i] if reg/mem[i] invalid, else = ~IntRes1[i]

4.1.5 Output Selection

Table 4-5. Output Selection

Imm8[6]	Operation	Description
0b	Least significant index	The index returned to ECX is of the least significant set bit in IntRes2.
1b	Most significant index	The index returned to ECX is of the most significant set bit in IntRes2.

For PCMPESTRI/PCMPISTRI, the Imm8 Control Byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

Table 4-6. Output Selection

Imm8[6]	Operation	Description
0b	Bit mask	IntRes2 is returned as the mask to the least significant bits of XMM0 with zero extension to 128 bits.
1b	Byte/word mask	IntRes2 is expanded into a byte/word mask (based on imm8[1]) and placed in XMM0. The expansion is performed by replicating each bit into all of the bits of the byte/word of the same index.

Specifically for PCMPESTRM/PCMPISTRM, the Imm8 Control Byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

4.1.6 Valid/Invalid Override of Comparisons

PCMPxSTRx instructions allow for the possibility that an end-of-string (EOS) situation may occur within the 128-bit packed data value (see the instruction descriptions below for details). Any data elements on either source that are determined to be past the EOS are considered to be invalid, and the treatment of invalid data within a comparison pair varies depending on the aggregation function being performed.

In general, the individual comparison result for each element pair BoolRes[i.j] can be forced true or false if one or more elements in the pair are invalid. See Table 4-7.

Table 4-7. Comparison Result for Each Element Pair BoolRes[i.j]

xmm1 byte/ word	xmm2/ m128 byte/word	Imm8[3:2] = 00b (equal any)	Imm8[3:2] = 01b (ranges)	Imm8[3:2] = 10b (equal each)	Imm8[3:2] = 11b (equal ordered)
Invalid	Invalid	Force false	Force false	Force true	Force true
Invalid	Valid	Force false	Force false	Force false	Force true
Valid	Invalid	Force false	Force false	Force false	Force false
Valid	Valid	Do not force	Do not force	Do not force	Do not force

4.1.7 Summary of Im8 Control byte

Table 4-8. Summary of Imm8 Control Byte

Imm8	Description
-------0b	128-bit sources treated as 16 packed bytes.
-------1b	128-bit sources treated as 8 packed words.
------0-b	Packed bytes/words are unsigned.
------1-b	Packed bytes/words are signed.
----00--b	Mode is equal any.
----01--b	Mode is ranges.
----10--b	Mode is equal each.
----11--b	Mode is equal ordered.
---0----b	IntRes1 is unmodified.
---1----b	IntRes1 is negated (1's complement).
--0-----b	Negation of IntRes1 is for all 16 (8) bits.
--1-----b	Negation of IntRes1 is masked by reg/mem validity.
-0------b	Index of the least significant, set, bit is used (regardless of corresponding input element validity). IntRes2 is returned in least significant bits of XMM0.
-1------b	Index of the most significant, set, bit is used (regardless of corresponding input element validity). Each bit of IntRes2 is expanded to byte/word.
0-------b	This bit currently has no defined effect, should be 0.
1-------b	This bit currently has no defined effect, should be 0.

4.1.8 Diagram Comparison and Aggregation Process

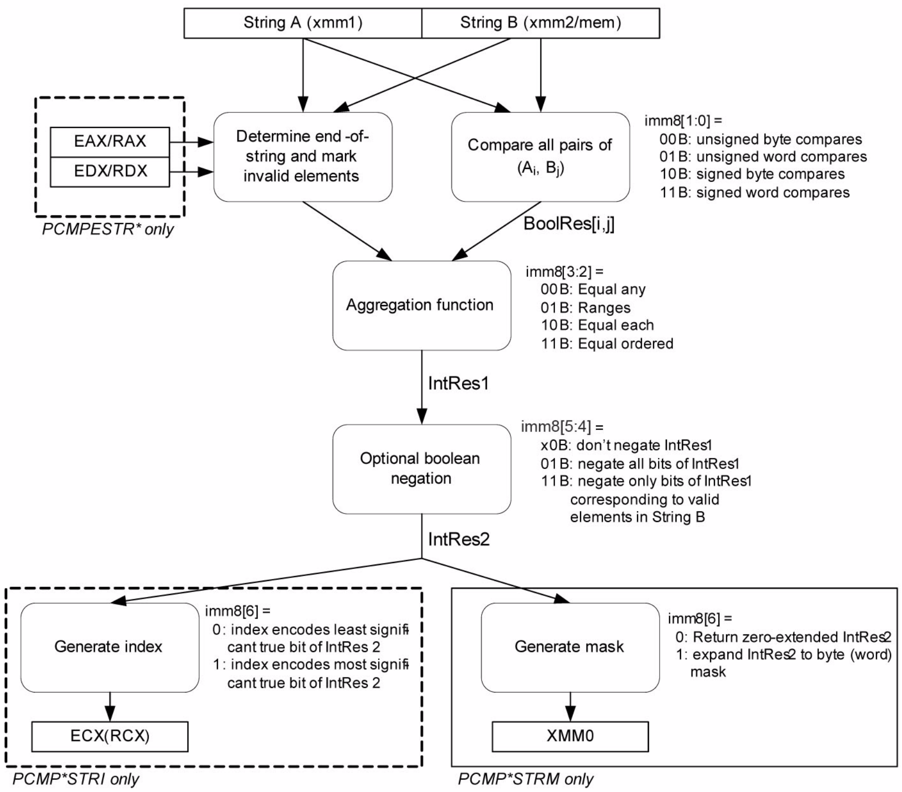

Figure 4-1. Operation of PCMPSTRx and PCMPESTRx

4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1 and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- f0(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.

  f0(B,C,D) ← (B AND C) XOR ((NOT(B) AND D)

- f1(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.

  f1(B,C,D) ← B XOR C XOR D

- f2(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.

  f2(B,C,D) ← (B AND C) XOR (B AND D) XOR (C AND D)

- f3(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as f1().

  f3(B,C,D) ← B XOR C XOR D

- Ch(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).

  Ch(E,F,G) ← (E AND F) XOR ((NOT E) AND G)

- Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).

  Maj(A,B,C) ← (A AND B) XOR (A AND C) XOR (B AND C)

ROR is rotate right operation

(A ROR N) ← A[N-1:0] || A[Width-1:N]

ROL is rotate left operation

(A ROL N) ← A ROR (Width-N)

SHR is the right shift operation

(A SHR N) ← ZEROES[N-1:0] || A[Width-1:N]

- Σ_0( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.

  Σ_0(A) ← (A ROR 2) XOR (A ROR 13) XOR (A ROR 22)

- Σ_1( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.

  Σ_1(E) ← (E ROR 6) XOR (E ROR 11) XOR (E ROR 25)

- σ_0( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.

  σ_0(W) ← (W ROR 7) XOR (W ROR 18) XOR (W SHR 3)

- σ_1( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.

  σ_1(W) ← (W ROR 17) XOR (W ROR 19) XOR (W SHR 10)

- K_i: SHA1 Constants dependent on immediate i.

  K0 = 0x5A827999

  K1 = 0x6ED9EBA1

  K2 = 0X8F1BBCDC

  K3 = 0xCA62C1D6

4.3 INSTRUCTIONS (M-U)

Chapter 4 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (M-U). See also: Chapter 3, “Instruction Set Reference, A-L,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, and Chapter 5, “Instruction Set Reference, V-Z,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*.

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /*r* MASKMOVDQU *xmm1, xmm2*	RM	V/V	SSE2	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU *xmm1, xmm2*	RM	V/V	AVX	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding[1]

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

Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

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

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

If VMASKMOVDQU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

1. ModRM.MOD = 011B required

Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DI/EDI +15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L= 1
	If VEX.vvvv ≠ 1111B.

MASKMOVQ—Store Selected Bytes of Quadword

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F F7 /r MASKMOVQ *mm1, mm2*	RM	Valid	Valid	Selectively write bytes from *mm1* to memory location using the byte mask in *mm2.* The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding

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

Description

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI +15] ← SRC[63:56] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmove_si64(__m64d, __m64n, char * p)
```

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

MAXPD—Maximum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN and OR.

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 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 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

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPD (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] ← MAX(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMAXPD (VEX.256 encoded version)

```
DEST[63:0] ←MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] ←MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] ←MAX(SRC1[255:192], SRC2[255:192])
DEST[MAXVL-1:256] ←0
```

VMAXPD (VEX.128 encoded version)

```
DEST[63:0] ←MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←MAX(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] ←0
```

MAXPD (128-bit Legacy SSE version)

DEST[63:0] ←MAX(DEST[63:0], SRC[63:0])
DEST[127:64] ←MAX(DEST[127:64], SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD __m512d _mm512_max_pd(__m512d a, __m512d b);
VMAXPD __m512d _mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b,);
VMAXPD __m512d _mm512_maskz_max_pd(__mmask8 k, __m512d a, __m512d b);
VMAXPD __m512d _mm512_max_round_pd(__m512d a, __m512d b, int);
VMAXPD __m512d _mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m512d _mm512_maskz_max_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m256d _mm256_mask_max_pd(__m5256d s, __mmask8 k, __m256d a, __m256d b);
VMAXPD __m256d _mm256_maskz_max_pd(__mmask8 k, __m256d a, __m256d b);
VMAXPD __m128d _mm_mask_max_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMAXPD __m128d _mm_maskz_max_pd(__mmask8 k, __m128d a, __m128d b);
VMAXPD __m256d _mm256_max_pd (__m256d a, __m256d b);
(V)MAXPD __m128d _mm_max_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MAXPS—Maximum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F	Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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. 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 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

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPS (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*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMAXPS (VEX.256 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] ←MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] ←MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] ←MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] ←MAX(SRC1[255:224], SRC2[255:224])
DEST[MAXVL-1:256] ←0
```

VMAXPS (VEX.128 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MAX(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] ←0
```

MAXPS (128-bit Legacy SSE version)

DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[63:32] ←MAX(DEST[63:32], SRC[63:32])
DEST[95:64] ←MAX(DEST[95:64], SRC[95:64])
DEST[127:96] ←MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPS __m512 _mm512_max_ps(__m512 a, __m512 b);
VMAXPS __m512 _mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_maskz_max_ps(__mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_max_round_ps(__m512 a, __m512 b, int);
VMAXPS __m512 _mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_maskz_max_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 _mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 _mm256_maskz_max_ps(__mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 _mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 _mm_maskz_max_ps(__mmask8 k, __m128 a, __m128 b);
VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);
MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

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

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

VEX.128 and EVEX encoded version: 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 encoded version: The low quadword element of the destination operand is updated according to the writemask.

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

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXSD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VMAXSD (VEX.128 encoded version)

```
DEST[63:0] ←MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

MAXSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

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

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

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

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

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

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

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXSS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← MAX(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
```

VMAXSS (VEX.128 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

MAXSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSS __m128 _mm_max_round_ss( __m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MFENCE—Memory Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F0	MFENCE	ZO	Valid	Valid	Serializes load and store operations.

Instruction Operand Encoding

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

Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.[1] The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

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

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

Operation

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_mfence(void)
```

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

MINPD—Minimum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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 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 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

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPD (EVEX encoded version)

```
(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] ← MIN(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMINPD (VEX.256 encoded version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←MIN(SRC1[127:64], SRC2[127:64])
DEST[191:128] ←MIN(SRC1[191:128], SRC2[191:128])
DEST[255:192] ←MIN(SRC1[255:192], SRC2[255:192])
```

VMINPD (VEX.128 encoded version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←MIN(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] ←0
```

MINPD (128-bit Legacy SSE version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←MIN(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMINPD __m512d _mm512_min_pd(__m512d a, __m512d b);
VMINPD __m512d _mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_maskz_min_pd(__mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_min_round_pd(__m512d a, __m512d b, int);
VMINPD __m512d _mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d _mm512_maskz_min_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d _mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d _mm256_maskz_min_pd(__mmask8 k, __m256d a, __m256d b);
VMINPD __m128d _mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d _mm_maskz_min_pd(__mmask8 k, __m128d a, __m128d b);
VMINPD __m256d _mm256_min_pd (__m256d a, __m256d b);
MINPD __m128d _mm_min_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MINPS—Minimum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.128.0F.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.128.0F.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F	Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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. 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 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

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPS (EVEX encoded version)

```
(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) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[i+31:i])
            FI;
            ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMINPS (VEX.256 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] ←MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] ←MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] ←MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] ←MIN(SRC1[255:224], SRC2[255:224])
```

VMINPS (VEX.128 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] ←0
```

MINPS (128-bit Legacy SSE version)

DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMINPS __m512 _mm512_min_ps(__m512 a, __m512 b);
VMINPS __m512 _mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_maskz_min_ps(__mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_min_round_ps(__m512 a, __m512 b, int);
VMINPS __m512 _mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 _mm512_maskz_min_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 _mm256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 _mm256_maskz_min_ps(__mmask8 k, __m256 a, __m25 b);
VMINPS __m128 _mm_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 _mm_maskz_min_ps(__mmask8 k, __m128 a, __m128 b);
VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);
MINPS __m128 _mm_min_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.LIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

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

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

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

VEX.128 and EVEX encoded version: 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 encoded version: The low quadword element of the destination operand is updated according to the writemask.

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

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

MINSD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← MIN(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
```

MINSD (VEX.128 encoded version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

MINSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.LIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

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

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

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

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)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 encoded version: The low doubleword element of the destination operand is updated according to the writemask.

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

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

MINSS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← MIN(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
```

VMINSS (VEX.128 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

MINSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C8	MONITOR	ZO	Valid	Valid	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:RAX/EAX/AX.

Instruction Operand Encoding

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

Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The address is specified in RAX/EAX/AX and the size is based on the effective address size of the encoded instruction. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, “Multiple-Processor Management” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

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

Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If the value in EAX is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If ECX ≠ 0.
#SS(0)	If the value in EAX is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.
	If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
	If ECX ≠ 0.
#SS	If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.
	If RCX ≠ 0.
#SS(0)	If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If the current privilege level is not 0.
	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /*r*	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8.*
REX + 88 /*r*	MOV *r/m8****,*r8****	MR	Valid	N.E.	Move *r8* to *r/m8.*
89 /*r*	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16.*
89 /*r*	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32.*
REX.W + 89 /*r*	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64.*
8A /*r*	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8.*
REX + 8A /*r*	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8.*
8B /*r*	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16.*
8B /*r*	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32.*
REX.W + 8B /*r*	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64.*
8C /*r*	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16.*
REX.W + 8C /*r*	MOV *r16/r32/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r16/r32/r64/m16.*
REX.W + 8C /*r*	MOV *r64/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r64/m16.*
8E /*r*	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /*r*	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8.*
REX + B0+ *rb ib*	MOV *r8****, imm8*	OI	Valid	N.E.	Move *imm8* to *r8.*
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16.*
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32.*
REX.W + B8+ *rd io*	MOV *r64, imm64*	OI	Valid	N.E.	Move *imm64* to *r64.*
C6 */0 ib*	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8.*
REX + C6 */0 ib*	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8.*
C7 */0 iw*	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16.*
C7 */0 id*	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32.*
REX.W + C7 */0 id*	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64.*

NOTES:

* The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following "Description" section for further information).

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an event can be delivered. See Section 6.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.* Intel recommends that software use the LSS instruction to load the SS register and ESP together.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium and earlier processors.

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

Operation

```
DEST ← SRC;
```

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

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

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

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

Flags Affected

None

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If an attempt is made to load SS register with NULL segment selector when CPL = 3.
	If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the memory access to the descriptor table is non-canonical.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 20/*r* MOV *r32,* CR0–CR7	MR	N.E.	Valid	Move control register to *r32*.
0F 20/*r* MOV *r64,* CR0–CR7	MR	Valid	N.E.	Move extended control register to *r64*.
REX.R + 0F 20 /0 MOV *r64,* CR8	MR	Valid	N.E.	Move extended CR8 to *r64*.[1]
0F 22 /*r* MOV CR0–CR7, *r32*	RM	N.E.	Valid	Move *r32* to control register.
0F 22 /*r* MOV CR0–CR7, *r64*	RM	Valid	N.E.	Move *r64* to extended control register.
REX.R + 0F 22 /0 MOV CR8, *r64*	RM	Valid	N.E.	Move *r64* to extended CR8.[1]

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See "Control Registers" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. Use of the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

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

Operation

```
DEST ← SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write 1 to CR4.PCIDE.
	If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.

Real-Address Mode Exceptions

#GP	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write 1 to CR4.PCIDE.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.

Virtual-8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
	If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
	If an attempt is made to write a 1 to any reserved bit in CR3.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
	If an attempt is made to clear CR0.PG[bit 31].
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write a 1 to any reserved bit in CR8.
	If an attempt is made to write a 1 to any reserved bit in CR3.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.
	If the REX.R prefix is used to specify a register other than CR8.

MOV—Move to/from Debug Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 21/*r* MOV *r32,* DR0–DR7	MR	N.E.	Valid	Move debug register to *r32.*
0F 21/*r* MOV *r64,* DR0–DR7	MR	Valid	N.E.	Move extended debug register to *r64.*
0F 23 /*r* MOV DR0–DR7, *r32*	RM	N.E.	Valid	Move *r32* to debug register.
0F 23 /*r* MOV DR0–DR7, *r64*	RM	Valid	N.E.	Move *r64* to extended debug register.

Instruction Operand Encoding

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

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 17.2, "Debug Registers", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction's default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8–R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ← SRC;

FI;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
	If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

Real-Address Mode Exceptions

#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
	If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

Virtual-8086 Mode Exceptions

#GP(0)	The debug registers cannot be loaded or read when in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write a 1 to any of bits 63:32 in DR6.
	If an attempt is made to write a 1 to any of bits 63:32 in DR7.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
	If the LOCK prefix is used.
	If the REX.R prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 2, 4 or 8 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPD (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking

    FI;
ENDFOR;
```

VMOVAPD (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPD (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVAPD (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPD (VEX.128 encoded version, load - and register copy)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPD (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m512d _mm512_load_pd(void * m);
VMOVAPD __m512d _mm512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d _mm512_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm512_store_pd(void * d, __m512d a);
VMOVAPD void _mm512_mask_store_pd(void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d _mm256_mask_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d _mm256_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm256_mask_store_pd(void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d _mm_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d _mm_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm_mask_store_pd(void * d, __mmask8 k, __m128d a);
MOVAPD __m256d _mm256_load_pd (double * p);
MOVAPD void _mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d _mm_load_pd (double * p);
MOVAPD void _mm_store_pd(double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1.

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

MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX encoded versions, register-copy form)

```
(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] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPS (EVEX encoded versions, store-form)

```
(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]←
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVAPS (EVEX encoded versions, load-form)

```
(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] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVAPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPS (VEX.128 encoded version, load - and register copy)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPS (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVAPS __m512 _mm512_load_ps( void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps( __mmask16 k, void * m);
VMOVAPS void _mm512_store_ps( void * d, __m512 a);
VMOVAPS void _mm512_mask_store_ps( void * d, __mmask16 k, __m512 a);
VMOVAPS __m256 _mm256_mask_load_ps(__m256 a, __mmask8 k, void * s);
VMOVAPS __m256 _mm256_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm256_mask_store_ps( void * d, __mmask8 k, __m256 a);
VMOVAPS __m128 _mm_mask_load_ps(__m128 a, __mmask8 k, void * s);
VMOVAPS __m128 _mm_maskz_load_ps( __mmask8 k, void * s);
VMOVAPS void _mm_mask_store_ps( void * d, __mmask8 k, __m128 a);
MOVAPS __m256 _mm256_load_ps (float * p);
MOVAPS void _mm256_store_ps(float * p, __m256 a);
MOVAPS __m128 _mm_load_ps (float * p);
MOVAPS void _mm_store_ps(float * p, __m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

MOVBE—Move Data After Swapping Bytes

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 F0 /*r*	MOVBE *r16*, m16	RM	Valid	Valid	Reverse byte order in *m16 and move to r16.*
0F 38 F0 /*r*	MOVBE *r32*, m32	RM	Valid	Valid	Reverse byte order in *m32 and move to r32.*
REX.W + 0F 38 F0 /*r*	MOVBE *r64*, m64	RM	Valid	N.E.	Reverse byte order in *m64 and move to r64.*
0F 38 F1 /*r*	MOVBE *m16*, r16	MR	Valid	Valid	Reverse byte order in *r16 and move to m16.*
0F 38 F1 /*r*	MOVBE *m32*, r32	MR	Valid	Valid	Reverse byte order in *r32 and move to m32.*
REX.W + 0F 38 F1 /*r*	MOVBE *m64*, r64	MR	Valid	N.E.	Reverse byte order in *r64 and move to m64.*

Instruction Operand Encoding

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

Description

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

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

Operation

```
TEMP ← SRC

IF ( OperandSize = 16)
    THEN
        DEST[7:0] ← TEMP[15:8];
        DEST[15:8] ← TEMP[7:0];
    ELES IF   ( OperandSize = 32)
        DEST[7:0] ← TEMP[31:24];
        DEST[15:8] ← TEMP[23:16];
        DEST[23:16] ← TEMP[15:8];
        DEST[31:23] ← TEMP[7:0];
    ELSE IF ( OperandSize = 64)
        DEST[7:0] ← TEMP[63:56];
        DEST[15:8] ← TEMP[55:48];
        DEST[23:16] ← TEMP[47:40];
        DEST[31:24] ← TEMP[39:32];
        DEST[39:32] ← TEMP[31:24];
        DEST[47:40] ← TEMP[23:16];
        DEST[55:48] ← TEMP[15:8];
        DEST[63:56] ← TEMP[7:0];

FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.
	If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /r MOVD *mm, r/m32*	A	V/V	MMX	Move doubleword from *r/m32* to *mm*.
NP REX.W + 0F 6E /r MOVQ *mm, r/m64*	A	V/N.E.	MMX	Move quadword from *r/m64* to *mm*.
NP 0F 7E /r MOVD *r/m32, mm*	B	V/V	MMX	Move doubleword from *mm* to *r/m32*.
NP REX.W + 0F 7E /r MOVQ *r/m64, mm*	B	V/N.E.	MMX	Move quadword from *mm* to *r/m64*.
66 0F 6E /r MOVD *xmm, r/m32*	A	V/V	SSE2	Move doubleword from *r/m32* to *xmm*.
66 REX.W 0F 6E /r MOVQ *xmm, r/m64*	A	V/N.E.	SSE2	Move quadword from *r/m64* to *xmm*.
66 0F 7E /r MOVD *r/m32, xmm*	B	V/V	SSE2	Move doubleword from *xmm* register to *r/m32*.
66 REX.W 0F 7E /r MOVQ *r/m64, xmm*	B	V/N.E.	SSE2	Move quadword from *xmm* register to *r/m64*.
VEX.128.66.0F.W0 6E / VMOVD *xmm1, r32/m32*	A	V/V	AVX	Move doubleword from *r/m32* to *xmm1*.
VEX.128.66.0F.W1 6E /r VMOVQ *xmm1, r64/m64*	A	V/N.E[1].	AVX	Move quadword from *r/m64* to *xmm1*.
VEX.128.66.0F.W0 7E /r VMOVD *r32/m32, xmm1*	B	V/V	AVX	Move doubleword from *xmm1* register to *r/m32*.
VEX.128.66.0F.W1 7E /r VMOVQ *r64/m64, xmm1*	B	V/N.E[1].	AVX	Move quadword from *xmm1* register to *r/m64*.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	C	V/V	AVX512F	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	C	V/N.E.[1]	AVX512F	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	D	V/V	AVX512F	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	D	V/N.E.[1]	AVX512F	Move quadword from xmm1 register to r/m64.

NOTES:

1. 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	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

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

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

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

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (when destination operand is MMX technology register)

```
DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;
```

MOVD (when destination operand is XMM register)

```
DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000H;
DEST[MAXVL-1:128] (Unmodified)
```

MOVD (when source operand is MMX technology or XMM register)

DEST ← SRC[31:0];

VMOVD (VEX-encoded version when destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

MOVQ (when destination operand is XMM register)

DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[MAXVL-1:128] (Unmodified)

MOVQ (when destination operand is r/m64)

DEST[63:0] ← SRC[63:0];

MOVQ (when source operand is XMM register or r/m64)

DEST ← SRC[63:0];

VMOVQ (VEX-encoded version when destination is an XMM register)

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVD (EVEX-encoded version when destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

VMOVQ (EVEX-encoded version when destination is an XMM register)

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVD:	__m64 _mm_cvtsi32_si64 (int i)
MOVD:	int _mm_cvtsi64_si32 (__m64m)
MOVD:	__m128i _mm_cvtsi32_si128 (int a)
MOVD:	int _mm_cvtsi128_si32 (__m128i a)
MOVQ:	__int64 _mm_cvtsi128_si64(__m128i);
MOVQ:	__m128i _mm_cvtsi64_si128(__int64);
VMOVD	__m128i _mm_cvtsi32_si128(int);
VMOVD	int _mm_cvtsi128_si32(__m128i);
VMOVQ	__m128i _mm_cvtsi64_si128 (__int64);
VMOVQ	__int64 _mm_cvtsi128_si64(__m128i);
VMOVQ	__m128i _mm_loadl_epi64(__m128i * s);
VMOVQ	void _mm_storel_epi64(__m128i * d, __m128i s);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVDDUP—Replicate Double FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Move double-precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move even index double-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move even index double-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

For 256-bit or higher versions: Duplicates even-indexed double-precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double-precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

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

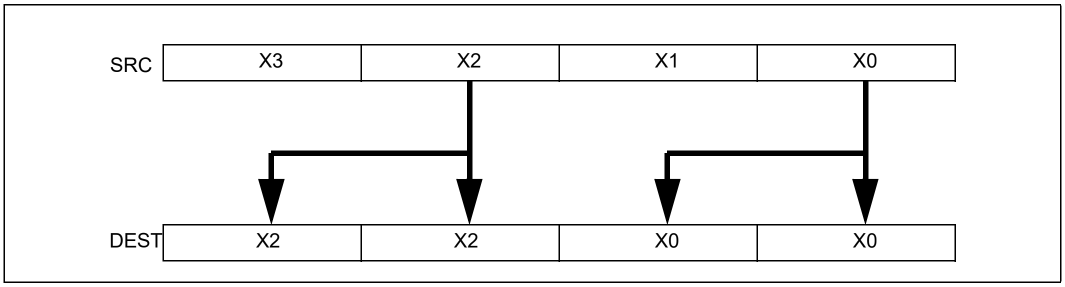

Figure 4-2. VMOVDDUP Operation

Operation

VMOVDDUP (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_SRC[63:0] ← SRC[63:0]
TMP_SRC[127:64] ← SRC[63:0]
IF VL >= 256
    TMP_SRC[191:128] ← SRC[191:128]
    TMP_SRC[255:192] ← SRC[191:128]
FI;
IF VL >= 512
    TMP_SRC[319:256] ← SRC[319:256]
    TMP_SRC[383:320] ← SRC[319:256]
    TMP_SRC[477:384] ← SRC[477:384]
    TMP_SRC[511:484] ← SRC[477:384]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0                ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDDUP (VEX.256 encoded version)

```
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[191:128] ←SRC[191:128]
DEST[255:192] ←SRC[191:128]
DEST[MAXVL-1:256] ←0
```

VMOVDDUP (VEX.128 encoded version)

```
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[MAXVL-1:128] ←0
```

MOVDDUP (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDDUP __m512d _mm512_movedup_pd(__m512d a);
VMOVDDUP __m512d _mm512_mask_movedup_pd(__m512d s, __mmask8 k, __m512d a);
VMOVDDUP __m512d _mm512_maskz_movedup_pd(__mmask8 k, __m512d a);
VMOVDDUP __m256d _mm256_mask_movedup_pd(__m256d s, __mmask8 k, __m256d a);
VMOVDDUP __m256d _mm256_maskz_movedup_pd(__mmask8 k, __m256d a);
VMOVDDUP __m128d _mm_mask_movedup_pd(__m128d s, __mmask8 k, __m128d a);
VMOVDDUP __m128d _mm_maskz_movedup_pd(__mmask8 k, __m128d a);
MOVDDUP __m256d _mm256_movedup_pd (__m256d a);
MOVDDUP __m128d _mm_movedup_pd (__m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E5NF.

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

MOVDIRI—Move Doubleword as Direct Store

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 F9 /r MOVDIRI m32, r32	A	V/V	MOVDIRI	Move doubleword from r32 to m32 using direct store.
NP REX.W + 0F 38 F9 /r MOVDIRI m64, r64	A	V/N.E.	MOVDIRI	Move quadword from r64 to m64 using direct store.

Instruction Operand Encoding[1]

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

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a direct-store operation. The source operand is a general purpose register. The destination operand is a 32-bit memory location. 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 summary chart at the beginning of this section for encoding data and limits.

The direct-store is implemented by using write combining (WC) memory type protocol for writing data. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store. Unlike stores with non-temporal hint that allow uncached (UC) and write-protected (WP) memory-type for the destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of the destination address memory type (including UC and WP types).

Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Because WC protocol used by direct-stores follows a weakly-ordered memory consistency model, a fencing operation using SFENCE or MFENCE should follow the MOVDIRI instruction to enforce ordering when needed.

Direct-stores issued by MOVDIRI to a destination aligned to a 4-byte boundary (8-byte boundary if used with REX.W prefix) guarantee 4-byte (8-byte with REX.W prefix) write-completion atomicity. This means that the data arrives at the destination in a single undivided 4-byte (or 8-byte) write transaction. If the destination is not aligned for the write size, the direct-stores issued by MOVDIRI are split and arrive at the destination in two parts. Each part of such split direct-store will not merge with younger stores but can arrive at the destination in either order. Availability of the MOVDIRI instruction is indicated by the presence of the CPUID feature flag MOVDIRI (bit 27 of the ECX register in leaf 07H, see "CPUID—CPU Identification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIRI void _directstoreu_u32(void *dst, uint32_t val)
MOVDIRI void _directstoreu_u64(void *dst, uint64_t val)

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

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.07H.0H:ECX.MOVDIRI[bit 27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF (fault-code)	For a page fault.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

MOVDIR64B—Move 64 Bytes as Direct Store

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F8 /r MOVDIR64B r16/r32/r64, m512	A	V/V	MOVDIR64B	Move 64-bytes as direct-store with guaranteed 64-byte write atomicity from the source memory operand address to destination memory address specified as offset to ES segment in the register operand.

Instruction Operand Encoding[1]

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

Description

Moves 64-bytes as direct-store with 64-byte write atomicity from source memory address to destination memory address. The source operand is a normal memory operand. The destination operand is a memory location specified in a general-purpose register. The register content is interpreted as an offset into ES segment without any segment override. In 64-bit mode, the register operand width is 64-bits (32-bits with 67H prefix). Outside of 64-bit mode, the register width is 32-bits when CS.D=1 (16-bits with 67H prefix), and 16-bits when CS.D=0 (32-bits with 67H prefix). MOVDIR64B requires the destination address to be 64-byte aligned. No alignment restriction is enforced for source operand.

MOVDIR64B reads 64-bytes from the source memory address and performs a 64-byte direct-store operation to the destination address. The load operation follows normal read ordering based on source address memory-type. The direct-store is implemented by using the write combining (WC) memory type protocol for writing data. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store.

Unlike stores with non-temporal hint which allow UC/WP memory-type for destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of destination address memory type (including UC/WP types). Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Because WC protocol used by direct-stores follow weakly-ordered memory consistency model, fencing operation using SFENCE or MFENCE should follow the MOVDIR64B instruction to enforce ordering when needed.

There is no atomicity guarantee provided for the 64-byte load operation from source address, and processor implementations may use multiple load operations to read the 64-bytes. The 64-byte direct-store issued by MOVDIR64B guarantees 64-byte write-completion atomicity. This means that the data arrives at the destination in a single undivided 64-byte write transaction.

Availability of the MOVDIR64B instruction is indicated by the presence of the CPUID feature flag MOVDIR64B (bit 28 of the ECX register in leaf 07H, see "CPUID—CPU Identification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIR64B void _movdir64b(void *dst, const void* src)

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

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
	If 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 address in destination (register) operand is not aligned to a 64-byte boundary.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
	If 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 memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
	If LOCK prefix is used.

MOVDQA,VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

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

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

Operation

VMOVDQA32 (EVEX encoded versions, register-copy form)

```
(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] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA32 (EVEX encoded versions, store-form)

```
(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]← SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA32 (EVEX encoded versions, load-form)

```
(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] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA64 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA64 (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA64 (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVDQA (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVDQA (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

VMOVDQA (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQA (128-bit store-form version)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA32 __m512i _mm512_load_epi32(void * sa);
VMOVDQA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i _mm512_maskz_load_epi32(__mmask16 k, void * sa);
VMOVDQA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i _mm256_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i _mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i _mm_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void _mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i _mm512_load_epi64(void * sa);
VMOVDQA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i _mm512_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i _mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i _mm256_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i _mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i _mm_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void _mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA void _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1.

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

MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/mm128	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

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

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

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

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

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

Operation

VMOVDQU8 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] ← 0           ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU8 (EVEX encoded versions, store-form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i]←
            SRC[i+7:i]
        ELSE *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU8 (EVEX encoded versions, load-form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] ← 0               ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU16 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU16 (EVEX encoded versions, store-form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i]←
            SRC[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU16 (EVEX encoded versions, load-form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU32 (EVEX encoded versions, register-copy form)

```
(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] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU32 (EVEX encoded versions, store-form)

```
(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]←
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU32 (EVEX encoded versions, load-form)

```
(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] ← SRC[i+31:i]
         ELSE
              IF *merging-masking*                   ; merging-masking
                   THEN *DEST[i+31:i] remains unchanged*
                   ELSE  DEST[i+31:i] ← 0            ; zeroing-masking
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU64 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
         THEN DEST[i+63:i] ← SRC[i+63:i]
         ELSE
              IF *merging-masking*                   ; merging-masking
                   THEN *DEST[i+63:i] remains unchanged*
                   ELSE  DEST[i+63:i] ← 0            ; zeroing-masking
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU64 (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
         THEN DEST[i+63:i]← SRC[i+63:i]
         ELSE *DEST[i+63:i] remains unchanged*         ; merging-masking

    FI;
ENDFOR;
```

VMOVDQU64 (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVDQU (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVDQU (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

VMOVDQU (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQU (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVDQU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVDQU16 __m512i _mm512_maskz_loadu_epi16( __mmask32 k, void * sa);
VMOVDQU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVDQU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVDQU16 __m256i _mm256_maskz_loadu_epi16( __mmask16 k, void * sa);
VMOVDQU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVDQU16 __m128i _mm_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVDQU16 __m128i _mm_maskz_loadu_epi16( __mmask8 k, void * sa);
VMOVDQU16 void _mm_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVDQU32 __m512i _mm512_loadu_epi32( void * sa);
VMOVDQU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQU32 __m512i _mm512_maskz_loadu_epi32( __mmask16 k, void * sa);
VMOVDQU32 void _mm512_storeu_epi32(void * d, __m512i a);
VMOVDQU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQU32 __m256i _mm256_maskz_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm256_storeu_epi32(void * d, __m256i a);
VMOVDQU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQU32 __m128i _mm_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQU32 __m128i _mm_maskz_loadu_epi32( __mmask8 k, void * sa);
```

VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64(void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8(__mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8(__mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8(__mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);
MOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4.nb.

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

MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F D6 /r	MOVDQ2Q *mm, xmm*	RM	Valid	Valid	Move low quadword from *xmm* to *mmx* register.

Instruction Operand Encoding

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

Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

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

Operation

```
DEST ← SRC[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q: __m64 _mm_movepi64_pi64 (__m128i a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

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

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit two-argument form)

```
DEST[63:0] ← SRC[127:64]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVHLPS (128-bit three-argument form - VEX & EVEX)

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

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point value from m64 to high quadword of xmm1.
VEX.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
EVEX.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	D	V/V	AVX512F	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	E	V/V	AVX512F	Move double-precision floating-point value from high quadword of xmm1 to m64.

Instruction Operand Encoding

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

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE load)

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

VMOVHPD (VEX.128 & EVEX encoded load)

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

VMOVHPD (store)

DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
EVEX.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	D	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	E	V/V	AVX512F	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE load)
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

VMOVHPS (VEX.128 and EVEX encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0

VMOVHPS (store)
DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128 _mm_loadh_pi (__m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

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

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLHPS (128-bit two-argument form)

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

VMOVLHPS (128-bit three-argument form - VEX & EVEX)

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

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point value from m64 to low quadword of xmm1.
VEX.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
EVEX.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	D	V/V	AVX512F	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	E	V/V	AVX512F	Move double-precision floating-point value from low quadword of xmm1 to m64.

Instruction Operand Encoding

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

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLPD (128-bit Legacy SSE load)

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

VMOVLPD (VEX.128 & EVEX encoded load)

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

VMOVLPD (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
EVEX.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	D	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	E	V/V	AVX512F	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE load)

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

VMOVLPS (VEX.128 & EVEX encoded load)

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

VMOVLPS (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /*r* MOVMSKPD *reg, xmm*	RM	V/V	SSE2	Extract 2-bit sign mask from *xmm* and store in *reg*. The upper bits of *r32* or *r64* are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD *reg, xmm2*	RM	V/V	AVX	Extract 2-bit sign mask from *xmm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD *reg, ymm2*	RM	V/V	AVX	Extract 4-bit sign mask from *ymm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.

Instruction Operand Encoding

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

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

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

Operation

(V)MOVMSKPD (128-bit versions)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
IF DEST = r32
    THEN DEST[31:2] ← 0;
    ELSE DEST[63:2] ← 0;
FI
```

VMOVMSKPD (VEX.256 encoded version)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
DEST[2] ← SRC[191]
DEST[3] ← SRC[255]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD: int _mm_movemask_pd (__m128d a)

VMOVMSKPD: _mm256_movemask_pd(__m256d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv ≠ 1111B.

MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 50 /*r* MOVMSKPS *reg, xmm*	RM	V/V	SSE	Extract 4-bit sign mask from *xmm* and store in *reg*. The upper bits of *r32* or *r64* are filled with zeros.
VEX.128.0F.WIG 50 /r VMOVMSKPS *reg, xmm2*	RM	V/V	AVX	Extract 4-bit sign mask from *xmm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.
VEX.256.0F.WIG 50 /r VMOVMSKPS *reg, ymm2*	RM	V/V	AVX	Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of *r32* or *r64* are zeroed.

Instruction Operand Encoding[1]

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

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

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

Operation

```
DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];

IF DEST = r32
    THEN DEST[31:4] ← ZeroExtend;
    ELSE DEST[63:4] ← ZeroExtend;
FI;
```

1. ModRM.MOD = 011B required

(V)MOVMSKPS (128-bit version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

VMOVMSKPS (VEX.256 encoded version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_movemask_ps(__m128 a)

int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	A	V/V	AVX2	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	AVX512VL AVX512F	Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	AVX512VL AVX512F	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of

a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor's implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementa-

1. ModRM.MOD != 011B

tion may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

Operation

MOVNTDQA (128bit- Legacy SSE form)
DEST ←SRC
DEST[MAXVL-1:128] (Unmodified)

VMOVNTDQA (VEX.128 and EVEX.128 encoded form)
DEST ← SRC
DEST[MAXVL-1:128] ← 0

VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0

VMOVNTDQA (EVEX.512 encoded form)
DEST[511:0] ← SRC[511:0]
DEST[MAXVL-1:512] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA __m512i _mm512_stream_load_si512(__m512i const* p);
MOVNTDQA __m128i _mm_stream_load_si128 (const __m128i *p);
VMOVNTDQA __m256i _mm256_stream_load_si256 (__m256i const* p);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1;

EVEX-encoded instruction, see Exceptions Type E1NF.

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

MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	B	V/V	AVX512F	Move packed integer values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTDQ(EVEX encoded versions)

```
VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0
```

1. ModRM.MOD != 011B

MOVNTDQ (Legacy and VEX versions)

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);

VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);

MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1NF.

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

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C3 /*r*	MOVNTI *m32, r32*	MR	Valid	Valid	Move doubleword from *r32* to *m32* using non-temporal hint.
NP REX.W + 0F C3 /*r*	MOVNTI *m64, r64*	MR	Valid	N.E.	Move quadword from *r64* to *m64* using non-temporal hint.

Instruction Operand Encoding

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

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI: void _mm_stream_si32 (int *p, int a)

MOVNTI: void _mm_stream_si64(__int64 *p, __int64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	B	V/V	AVX512F	Move packed double-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTPD (EVEX encoded versions)

```
VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0
```

1. ModRM.MOD != 011B

MOVNTPD (Legacy and VEX versions)

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1NF.

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

MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed single-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed single-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	B	V/V	AVX512F	Move packed single-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

VMOVNTPS (EVEX encoded versions)

```
VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0
```

1. ModRM.MOD != 011B

MOVNTPS

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

EVEX-encoded instruction, see Exceptions Type E1NF.

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

MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F E7 /*r*	MOVNTQ *m64, mm*	MR	Valid	Valid	Move quadword from *mm* to *m64* using non-temporal hint.

Instruction Operand Encoding

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

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ: void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /*r* MOVQ *mm, mm/m64*	A	V/V	MMX	Move quadword from *mm/m64* to *mm*.
NP 0F 7F /*r* MOVQ *mm/m64, mm*	B	V/V	MMX	Move quadword from *mm* to *mm/m64*.
F3 0F 7E /r MOVQ *xmm1*, *xmm2/m64*	A	V/V	SSE2	Move quadword from *xmm2/mem64* to *xmm1*.
VEX.128.F3.0F.WIG 7E /r VMOVQ *xmm1, xmm2/m64*	A	V/V	AVX	Move quadword from *xmm2* to *xmm1*.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	C	V/V	AVX512F	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ *xmm2/m64, xmm1*	B	V/V	SSE2	Move quadword from *xmm1* to *xmm2/mem64*.
VEX.128.66.0F.WIG D6 /r VMOVQ *xmm1/m64, xmm2*	B	V/V	AVX	Move quadword from *xmm2* register to *xmm1/m64*.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	D	V/V	AVX512F	Move quadword from xmm2 register to xmm1/m64.

Instruction Operand Encoding

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

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVQ instruction when operating on MMX technology registers and memory locations

DEST ← SRC;

MOVQ instruction when source and destination operands are XMM registers

DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;

MOVQ instruction when source operand is XMM register and destination
operand is memory location:

DEST ← SRC[63:0];

MOVQ instruction when source operand is memory location and destination
operand is XMM register:

DEST[63:0] ← SRC;
DEST[127:64] ← 0000000000000000H;

VMOVQ (VEX.128.F3.0F 7E) with XMM register source and destination

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with XMM register source and destination

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (D6 - EVEX encoded version) with XMM register source and destination

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (7E) with memory source

DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVQ (7E - EVEX encoded version) with memory source

DEST[63:0] ← SRC[63:0]
DEST[:MAXVL-1:64] ← 0

VMOVQ (D6) with memory dest

DEST[63:0] ← SRC2[63:0]

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ __m128i _mm_loadu_si64(void * s);
VMOVQ void _mm_storeu_si64(void * d, __m128i s);

MOVQ m128i _mm_move_epi64(__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F D6 /r	MOVQ2DQ *xmm, mm*	RM	Valid	Valid	Move quadword from *mmx* to low quadword of *xmm.*

Instruction Operand Encoding

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

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

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

Operation

```
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 00000000000000000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ: __128i _mm_movpi64_epi64 (__m64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A4	MOVS *m8, m8*	ZO	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R|E)SI to (R|E)DI.
A5	MOVS *m16, m16*	ZO	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R|E)SI to (R|E)DI.
A5	MOVS *m32, m32*	ZO	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R|E)SI to (R|E)DI.
REX.W + A5	MOVS *m64, m64*	ZO	Valid	N.E.	Move qword from address (R|E)SI to (R|E)DI.
A4	MOVSB	ZO	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R|E)SI to (R|E)DI.
A5	MOVSW	ZO	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R|E)SI to (R|E)DI.
A5	MOVSD	ZO	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R|E)SI to (R|E)DI.
REX.W + A5	MOVSQ	ZO	Valid	N.E.	Move qword from address (R|E)SI to (R|E)DI.

Instruction Operand Encoding

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

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incremented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

NOTE

> To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with MOVS and MOVSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← SRC;

Non-64-bit Mode:

IF (Byte move)
    THEN IF DF = 0
        THEN
            (E)SI ← (E)SI + 1;
            (E)DI ← (E)DI + 1;
        ELSE
            (E)SI ← (E)SI – 1;
            (E)DI ← (E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (E)SI ← (E)SI + 2;
            (E)DI ← (E)DI + 2;
            FI;
        ELSE
            (E)SI ← (E)SI – 2;
            (E)DI ← (E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (E)SI ← (E)SI + 4;
            (E)DI ← (E)DI + 4;
            FI;
        ELSE
            (E)SI ← (E)SI – 4;
            (E)DI ← (E)DI – 4;
        FI;
FI;
64-bit Mode:

IF (Byte move)
    THEN IF DF = 0
        THEN
```

```
            (R|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Load scalar double-precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	C	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double-precision floating-point value from m64 to xmm1 register.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Store scalar double-precision floating-point value from xmm1 register to m64.
EVEX.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	F	V/V	AVX512F	Load scalar double-precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.LIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	G	V/V	AVX512F	Store scalar double-precision floating-point value from xmm1 register to m64 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	NA	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double-precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation

VMOVSD (EVEX.LIG.F2.0F 10 /r: VMOVSD xmm1, m64 with support for 32 registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC[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[MAXVL-1:64] ← 0
```

VMOVSD (EVEX.LIG.F2.0F 11 /r: VMOVSD m64, xmm1 with support for 32 registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC[63:0]
    ELSE    *DEST[63:0] remains unchanged*          ; merging-masking
FI;
```

VMOVSD (EVEX.LIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
IF k1[0] or *no writemask*
    THEN    DEST[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
```

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)
DEST[63:0] ←SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

VMOVSD (VEX.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ←SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ←SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] ←SRC[63:0]
DEST[MAXVL-1:64] ←0

MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)
DEST[63:0] ←SRC[63:0]

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←0
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd(__mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd(__mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

MOVSHDUP—Replicate Single FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates odd-indexed single-precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

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

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

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

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

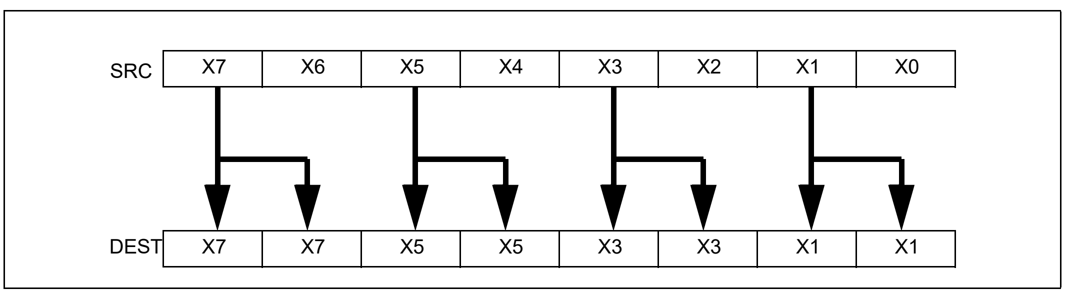

Figure 4-3. MOVSHDUP Operation

Operation

VMOVSHDUP (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] ← SRC[63:32]
TMP_SRC[63:32] ← SRC[63:32]
TMP_SRC[95:64] ← SRC[127:96]
TMP_SRC[127:96] ← SRC[127:96]
IF VL >= 256
    TMP_SRC[159:128] ← SRC[191:160]
    TMP_SRC[191:160] ← SRC[191:160]
    TMP_SRC[223:192] ← SRC[255:224]
    TMP_SRC[255:224] ← SRC[255:224]
FI;
IF VL >= 512
    TMP_SRC[287:256] ← SRC[319:288]
    TMP_SRC[319:288] ← SRC[319:288]
    TMP_SRC[351:320] ← SRC[383:352]
    TMP_SRC[383:352] ← SRC[383:352]
    TMP_SRC[415:384] ← SRC[447:416]
    TMP_SRC[447:416] ← SRC[447:416]
    TMP_SRC[479:448] ← SRC[511:480]
    TMP_SRC[511:480] ← SRC[511:480]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_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
```

VMOVSHDUP (VEX.256 encoded version)

```
DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[159:128] ← SRC[191:160]
DEST[191:160] ← SRC[191:160]
DEST[223:192] ← SRC[255:224]
DEST[255:224] ← SRC[255:224]
DEST[MAXVL-1:256] ← 0
```

VMOVSHDUP (VEX.128 encoded version)

```
DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[MAXVL-1:128] ← 0
```

MOVSHDUP (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSHDUP __m512 _mm512_movehdup_ps( __m512 a);
VMOVSHDUP __m512 _mm512_mask_movehdup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSHDUP __m512 _mm512_maskz_movehdup_ps( __mmask16 k, __m512 a);
VMOVSHDUP __m256 _mm256_mask_movehdup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_maskz_movehdup_ps( __mmask8 k, __m256 a);
VMOVSHDUP __m128 _mm_mask_movehdup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm_maskz_movehdup_ps( __mmask8 k, __m128 a);
VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

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

MOVSLDUP—Replicate Single FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move even index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

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

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

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

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

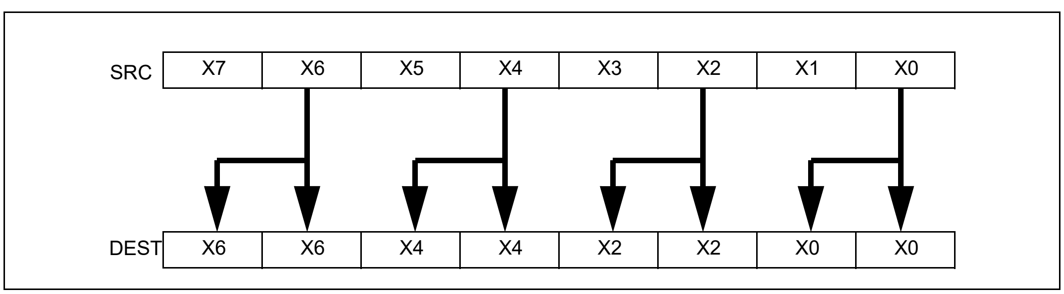

Figure 4-4. MOVSLDUP Operation

Operation

VMOVSLDUP (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] ← SRC[31:0]
TMP_SRC[63:32] ← SRC[31:0]
TMP_SRC[95:64] ← SRC[95:64]
TMP_SRC[127:96] ← SRC[95:64]
IF VL >= 256
    TMP_SRC[159:128] ← SRC[159:128]
    TMP_SRC[191:160] ← SRC[159:128]
    TMP_SRC[223:192] ← SRC[223:192]
    TMP_SRC[255:224] ← SRC[223:192]
FI;
IF VL >= 512
    TMP_SRC[287:256] ← SRC[287:256]
    TMP_SRC[319:288] ← SRC[287:256]
    TMP_SRC[351:320] ← SRC[351:320]
    TMP_SRC[383:352] ← SRC[351:320]
    TMP_SRC[415:384] ← SRC[415:384]
    TMP_SRC[447:416] ← SRC[415:384]
    TMP_SRC[479:448] ← SRC[479:448]
    TMP_SRC[511:480] ← SRC[479:448]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_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
```

VMOVSLDUP (VEX.256 encoded version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[159:128] ← SRC[159:128]
DEST[191:160] ← SRC[159:128]
DEST[223:192] ← SRC[223:192]
DEST[255:224] ← SRC[223:192]
DEST[MAXVL-1:256] ← 0
```

VMOVSLDUP (VEX.128 encoded version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[MAXVL-1:128] ← 0
```

MOVSLDUP (128-bit Legacy SSE version)
DEST[31:0] ←SRC[31:0]
DEST[63:32] ←SRC[31:0]
DEST[95:64] ←SRC[95:64]
DEST[127:96] ←SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSLDUP __m512 _mm512_moveldup_ps(__m512 a);
VMOVSLDUP __m512 _mm512_mask_moveldup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSLDUP __m512 _mm512_maskz_moveldup_ps(__mmask16 k, __m512 a);
VMOVSLDUP __m256 _mm256_mask_moveldup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSLDUP __m256 _mm256_maskz_moveldup_ps(__mmask8 k, __m256 a);
VMOVSLDUP __m128 _mm_mask_moveldup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSLDUP __m128 _mm_maskz_moveldup_ps(__mmask8 k, __m128 a);
VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

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

MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single-precision floating-point value from xmm1 register to m32.
EVEX.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	F	V/V	AVX512F	Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.LIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	G	V/V	AVX512F	Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	NA	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single-precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

VMOVSS (EVEX.LIG.F3.0F.W0 11 /r when the source operand is memory and the destination is an XMM register)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC[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[MAXVL-1:32] ← 0
```

VMOVSS (EVEX.LIG.F3.0F.W0 10 /r when the source operand is an XMM register and the destination is memory)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC[31:0]
    ELSE    *DEST[31:0] remains unchanged*          ; merging-masking
FI;
```

VMOVSS (EVEX.LIG.F3.0F.W0 10/11 /r where the source and destination are XMM registers)

```
IF k1[0] or *no writemask*
    THEN      DEST[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
```

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)

```
DEST[31:0] ←SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

VMOVSS (VEX.128.F3.0F 11 /r where the destination is an XMM register)

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

VMOVSS (VEX.128.F3.0F 10 /r where the source and destination are XMM registers)

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

VMOVSS (VEX.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)

```
DEST[31:0] ←SRC[31:0]
DEST[MAXVL-1:32] ←0
```

MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)

```
DEST[31:0] ←SRC[31:0]
```

MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)

```
DEST[31:0] ←SRC[31:0]
DEST[127:32] ←0
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss( __mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss( __mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /*r*	MOVSX *r16, r/m8*	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /*r*	MOVSX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + 0F BE /*r*	MOVSX *r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.
0F BF /*r*	MOVSX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + 0F BF /*r*	MOVSX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /*r**	MOVSXD *r16, r/m16*	RM	Valid	Valid	Move word to word with sign-extension.
63 /*r**	MOVSXD *r32, r/m32*	RM	Valid	Valid	Move doubleword to doubleword with sign-extension.
REX.W + 63 /*r*	MOVSXD *r64, r/m32*	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

* The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

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

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

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

Operation

```
DEST ← SignExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double-precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

<u>128-bit versions</u>:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

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

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPD (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPD (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*       ; merging-masking

    FI;
ENDFOR;
```

VMOVUPD (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPD (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVUPD (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVUPD (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVUPD (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPD (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPD __m512d _mm512_loadu_pd( void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd( __mmask8 k, void * s);
VMOVUPD void _mm512_storeu_pd( void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd( void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d s, __mmask8 k, void * m);
VMOVUPD __m256d _mm256_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm256_mask_storeu_pd( void * d, __mmask8 k, __m256d a);
VMOVUPD __m128d _mm_mask_loadu_pd(__m128d s, __mmask8 k, void * m);
VMOVUPD __m128d _mm_maskz_loadu_pd( __mmask8 k, void * m);
VMOVUPD void _mm_mask_storeu_pd( void * d, __mmask8 k, __m128d a);
MOVUPD __m256d _mm256_loadu_pd (double * p);
MOVUPD void _mm256_storeu_pd( double *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (double * p);
MOVUPD void _mm_storeu_pd( double *p, __m128d a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
NP 0F 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
VEX.128.0F.WIG 11 /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F.WIG 10 /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.
VEX.256.0F.WIG 11 /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.0F.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

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

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX encoded versions, register-copy form)

```
(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] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*              ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0        ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPS (EVEX encoded versions, store-form)

```
(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]← SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVUPS (EVEX encoded versions, load-form)

```
(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] ← SRC[i+31:i]
         ELSE
              IF *merging-masking*                  ; merging-masking
                   THEN *DEST[i+31:i] remains unchanged*
                   ELSE  DEST[i+31:i] ← 0           ; zeroing-masking
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVUPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVUPS (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVUPS (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPS (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPS __m512 _mm512_loadu_ps( void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps( __mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps( void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps( void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps( void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm_mask_storeu_ps( void * d, __mmask8 k, __m128 a);
MOVUPS __m256 _mm256_loadu_ps ( float * p);
MOVUPS void _mm256 _storeu_ps( float *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps ( float * p);
MOVUPS void _mm_storeu_ps( float *p, __m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies;

EVEX-encoded instruction, see Exceptions Type E4.nb.

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

MOVZX—Move with Zero-Extend

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B6 /*r*	MOVZX *r16, r/m8*	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /*r*	MOVZX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /*r*	MOVZX *r64, r/m8**	RM	Valid	N.E.	Move byte to quadword, zero-extension.
0F B7 /*r*	MOVZX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + 0F B7 /*r*	MOVZX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

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

Instruction Operand Encoding

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

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

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

Operation

```
DEST ← ZeroExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MPSADBW — Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm1* and *xmm2/m128* and writes the results in *xmm1*. Starting offsets within *xmm1* and *xmm2/m128* are determined by *imm8*.
VEX.128.66.0F3A.WIG 42 /r ib VMPSADBW *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm2* and *xmm3/m128* and writes the results in *xmm1*. Starting offsets within *xmm2* and *xmm3/m128* are determined by *imm8*.
VEX.256.66.0F3A.WIG 42 /r ib VMPSADBW *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm2* and *ymm3/m128* and writes the results in *ymm1*. Starting offsets within *ymm2* and *xmm3/m128* are determined by *imm8*.

Instruction Operand Encoding

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

Description

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

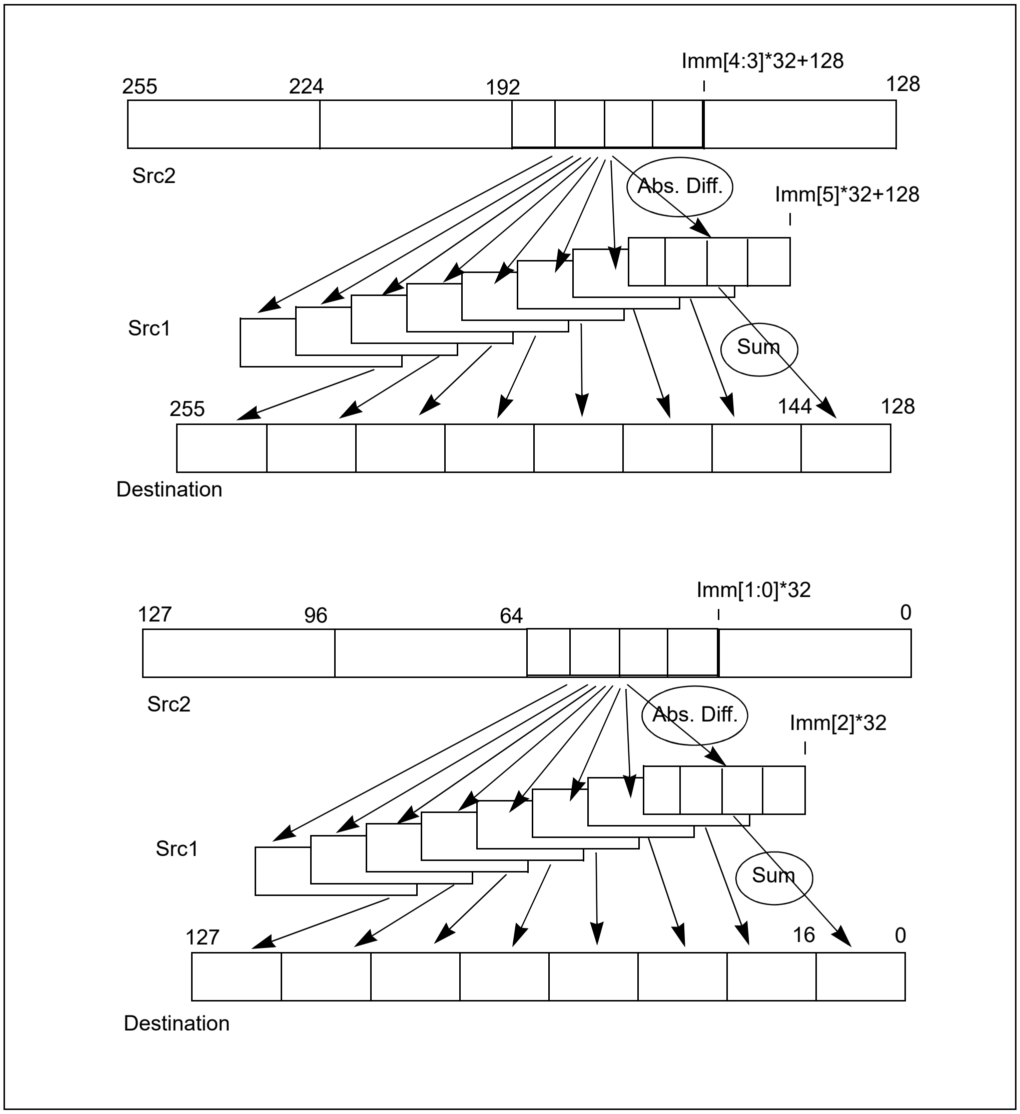

Figure 4-5. 256-bit VMPSADBW Operation

Operation

VMPSADBW (VEX.256 encoded version)

```
BLK2_OFFSET ← imm8[1:0]*32
BLK1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ←SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]
SRC2_BYTE0 ←SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

```
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

BLK2_OFFSET ← imm8[4:3]*32 + 128
BLK1_OFFSET ← imm8[5]*32 + 128
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 ←SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[143:128] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ←ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[159:144] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[175:160] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

```
TEMP0 ←ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[191:176] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[207:192] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[223:208] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[239:224] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[255:240] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

VMPSADBW (VEX.128 encoded version)

```
BLK2_OFFSET ← imm8[1:0]*32
BLK1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 ←SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]
```

```
TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] ← 0
```

MPSADBW (128-bit Legacy SSE version)

```
SRC_OFFSET ← imm8[1:0]*32
DEST_OFFSET ← imm8[2]*32
DEST_BYTE0 ← DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 ← DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 ← DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 ← DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 ← DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 ← DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 ← DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 ← DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 ← DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 ← DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 ← DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 ← SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYTE1 ← SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYTE2 ← SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYTE3 ← SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMP0 ← ABS( DEST_BYTE0 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE1 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE2 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE1 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE2 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE3 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE2 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE3 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE4 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE3 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE4 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE5 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE4 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE5 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE6 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

```
TEMP0 ← ABS( DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)MPSADBW: __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);

VMPSADBW: __m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);

Flags Affected

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL *r/m8*	M	Valid	Valid	Unsigned multiply (AX ← AL ∗ *r/m8*).
REX + F6 /4	MUL *r/m8**	M	Valid	N.E.	Unsigned multiply (AX ← AL ∗ *r/m8*).
F7 /4	MUL *r/m16*	M	Valid	Valid	Unsigned multiply (DX:AX ← AX ∗ *r/m16*).
F7 /4	MUL *r/m32*	M	Valid	Valid	Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*).
REX.W + F7 /4	MUL *r/m64*	M	Valid	N.E.	Unsigned multiply (RDX:RAX ← RAX ∗ *r/m64*).

NOTES:

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

Instruction Operand Encoding

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

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

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

Table 4-9. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX
Quadword	RAX	r/m64	RDX:RAX

Operation

```
IF (Byte operation)
    THEN
        AX ← AL ∗ SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX ← AX ∗ SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX ← EAX ∗ SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX ← RAX ∗ SRC;
        FI;
FI;
```

Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

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

Description

Multiply packed double-precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination 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 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. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

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 destination YMM 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

VMULPD (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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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
```

VMULPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64] * SRC2[127:64]
DEST[191:128] ←SRC1[191:128] * SRC2[191:128]
DEST[255:192] ←SRC1[255:192] * SRC2[255:192]
DEST[MAXVL-1:256] ←0;
.
```

VMULPD (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
```

MULPD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] * SRC[63:0]
DEST[127:64] ←DEST[127:64] * SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPD __m512d _mm512_mul_pd(__m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd(__mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd(__m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d _mm512_maskz_mul_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F	Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

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

Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination 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. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

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 destination YMM 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

VMULPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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
```

VMULPS (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;
```

VMULPS (VEX.128 encoded version)

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

MULPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);
MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

MULSD—Multiply Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1.
VEX.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.
EVEX.LIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	C	V/V	AVX512F	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.

Instruction Operand Encoding

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

Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in 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 operand and the destination operands 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 destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

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

Operation

VMULSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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;
ENDFOR
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VMULSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

MULSD (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

```
VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1.
VEX.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.
EVEX.LIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	C	V/V	AVX512F	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.

Instruction Operand Encoding

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

Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

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

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

Operation

VMULSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VMULSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

MULSS (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

```
VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MULX — Unsigned Multiply Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F38.W0 F6 /r MULX *r32a, r32b, r/m32*	RVM	V/V	BMI2	Unsigned multiply of *r/m32* with EDX without affecting arithmetic flags.
VEX.LZ.F2.0F38.W1 F6 /r MULX *r64a, r64b, r/m64*	RVM	V/N.E.	BMI2	Unsigned multiply of *r/m64* with RDX without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

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

Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
        DEST2 ← (SRC1*SRC2)[63:0];
        DEST1 ← (SRC1*SRC2)[127:64];
FI
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);

unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 13.

MWAIT—Monitor Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C9	MWAIT	ZO	Valid	Valid	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding

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

Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

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

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized "waiting" states and will select among those states based on the hint argument.

Table 4-10 describes the meaning of ECX and EAX registers for MWAIT extensions.

Table 4-10. MWAIT Extension Register (ECX)

Bits	Description
0	Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF=0). May be set only if CPUID.05H:ECX[bit 1] = 1.
31: 1	Reserved

Table 4-11. MWAIT Hints Register (EAX)

Bits	Description
3 : 0	Sub C-state within a C-state, indicated by bits [7:4]
7 : 4	Target C-state* Value of 0 means C1; 1 means C2 and so on Value of 01111B means C0 Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states
31: 8	Reserved

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 14, "Power and Thermal Management," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Operation

```
(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension
MWAIT EAX, ECX *)
{
WHILE ( ("Monitor Hardware is in armed state")) {
    implementation_dependent_optimized_state(EAX, ECX); }
Set the state of Monitor Hardware as triggered;
}
```

Intel C/C++ Compiler Intrinsic Equivalent

MWAIT: void _mm_mwait(unsigned extensions, unsigned hints)

Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If ECX[31:1] ≠ 0.
	If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.
	If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If ECX[31:1] ≠ 0.
	If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If RCX[63:1] ≠ 0.
	If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If the current privilege level is not 0.
	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

NEG—Two's Complement Negation

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

NOTES:

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

Instruction Operand Encoding

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

Description

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

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

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

Operation

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

Flags Affected

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

Protected Mode Exceptions

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

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

NOP—No Operation

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

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

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

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

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

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

Flags Affected

None

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

NOT—One's Complement Negation

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

NOTES:

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

Instruction Operand Encoding

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

Description

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

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

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

Operation

```
DEST ← NOT DEST;
```

Flags Affected

None

Protected Mode Exceptions

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

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

OR—Logical Inclusive OR

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

NOTES:

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

Instruction Operand Encoding

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

Description

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

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

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

Operation

DEST ← DEST OR SRC;

Flags Affected

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

Protected Mode Exceptions

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

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical OR 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	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR 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 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 register destination are unmodified.

Operation

VORPD (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 OR SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VORPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VORPD (VEX.128 encoded version)

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

ORPD (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VORPD __m512d _mm512_or_pd ( __m512d a, __m512d  b);
VORPD __m512d _mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d _mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);
VORPD __m256d _mm256_mask_or_pd (__m256d s, ___mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_maskz_or_pd (__mmask8 k, __m256d a, __m256d b);
VORPD __m128d _mm_mask_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_maskz_or_pd (__mmask8 k, __m128d a, __m128d b);
VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);
ORPD __m128d _mm_or_pd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical OR 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	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR 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 register destination are unmodified.

Operation

VORPS (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*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VORPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAXVL-1:256] ← 0
```

VORPS (VEX.128 encoded version)

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

ORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, ___mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

OUT—Output to Port

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

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	NA	NA	NA
ZO	NA	NA	NA	NA

Description

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

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

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

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

Operation

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

Flags Affected

None

Protected Mode Exceptions

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

64-Bit Mode Exceptions

Same as protected mode exceptions.

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

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

NOTES:

* See IA-32 Architecture Compatibility section below.

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

Instruction Operand Encoding

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

Description

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

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

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

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

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

Operation

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

Byte transfer:
    IF 64-bit mode
        Then
            IF 64-Bit Address Size
                THEN
                    IF DF = 0
                        THEN RSI ← RSI RSI + 1;
                        ELSE RSI ← RSI or – 1;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI ← ESI + 1;
                        ELSE    ESI ← ESI – 1;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN    (E)SI ← (E)SI + 1;
                ELSE (E)SI ← (E)SI – 1;
            FI;
    FI;
Word transfer:
    IF 64-bit mode
```

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

Flags Affected

None

Protected Mode Exceptions

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

64-Bit Mode Exceptions

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

PABSB/PABSW/PABSD/PABSQ — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r[1] PABSB *mm1, mm2/m64*	A	V/V	SSSE3	Compute the absolute value of bytes in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1C /r PABSB *xmm1, xmm2/m128*	A	V/V	SSSE3	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
NP 0F 38 1D /r[1] PABSW *mm1, mm2/m64*	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1D /r PABSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
NP 0F 38 1E /r[1] PABSD *mm1, mm2/m64*	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1E /r PABSD *xmm1, xmm2/m128*	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1.*
VEX.128.66.0F38.WIG 1C /r VPABSB *xmm1, xmm2/m128*	A	V/V	AVX	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1D /r VPABSW *xmm1, xmm2/m128*	A	V/V	AVX	Compute the absolute value of 16- bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1E /r VPABSD *xmm1, xmm2/m128*	A	V/V	AVX	Compute the absolute value of 32- bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.256.66.0F38.WIG 1C /r VPABSB *ymm1, ymm2/m256*	A	V/V	AVX2	Compute the absolute value of bytes in *ymm2/m256* and store UNSIGNED result in *ymm1*.
VEX.256.66.0F38.WIG 1D /r VPABSW *ymm1, ymm2/m256*	A	V/V	AVX2	Compute the absolute value of 16-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.
VEX.256.66.0F38.WIG 1E /r VPABSD *ymm1, ymm2/m256*	A	V/V	AVX2	Compute the absolute value of 32-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.
EVEX.128.66.0F38.WIG 1C /r VPABSB xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1C /r VPABSB ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1C /r VPABSB zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW	Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.WIG 1D /r VPABSW xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.

EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1E /r VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	C	V/V	AVX512F	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	C	V/V	AVX512F	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The 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 register destination are zeroed.

VEX.128 encoded versions: 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 can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-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

PABSB with 128 bit operands:

```
    Unsigned DEST[7:0] ←ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120] ←ABS(SRC[127:120])
```

VPABSB with 128 bit operands:

```
    Unsigned DEST[7:0] ←ABS(SRC[7: 0])
    Repeat operation for 2nd through 15th bytes
    Unsigned DEST[127:120]←ABS(SRC[127:120])
```

VPABSB with 256 bit operands:

```
    Unsigned DEST[7:0]←ABS(SRC[7: 0])
    Repeat operation for 2nd through 31st bytes
    Unsigned DEST[255:248]←ABS(SRC[255:248])
```

VPABSB (EVEX encoded versions)

```
    (KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PABSW with 128 bit operands:

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

VPABSW with 128 bit operands:

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

VPABSW with 256 bit operands:

```
    Unsigned DEST[15:0]←ABS(SRC[15:0])
    Repeat operation for 2nd through 15th 16-bit words
    Unsigned DEST[255:240] ←ABS(SRC[255:240])
```

VPABSW (EVEX encoded versions)

```
    (KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+15:i] ← ABS(SRC[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PABSD with 128 bit operands:

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

VPABSD with 128 bit operands:

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

VPABSD with 256 bit operands:

```
    Unsigned DEST[31:0] ←ABS(SRC[31:0])
    Repeat operation for 2nd through 7th 32-bit double words
    Unsigned DEST[255:224] ←ABS(SRC[255:224])
```

VPABSD (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*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+31:i] ← ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[i+31:i] ← ABS(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPABSQ (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 (SRC *is memory*)
                THEN
                    Unsigned DEST[i+63:i] ← ABS(SRC[63:0])
                ELSE
                    Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPABSB__m512i _mm512_abs_epi8 ( __m512i a)
VPABSW__m512i _mm512_abs_epi16 ( __m512i a)
VPABSB__m512i _mm512_mask_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i _mm512_mask_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i _mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW__m512i _mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB__m256i _mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW__m256i _mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB__m256i _mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW__m256i _mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB__m128i _mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW__m128i _mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB__m128i _mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW__m128i _mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i _mm256_maskz_abs_epi32( __mmask8 k, __m256i a);
VPABSD __m128i _mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i _mm_maskz_abs_epi32( __mmask8 k, __m128i a);
VPABSD  __m512i _mm512_abs_epi32( __m512i a);
VPABSD  __m512i _mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD  __m512i _mm512_maskz_abs_epi32( __mmask16 k, __m512i a);
VPABSQ __m512i _mm512_abs_epi64( __m512i a);
VPABSQ __m512i _mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i _mm512_maskz_abs_epi64( __mmask8 k, __m512i a);
VPABSQ __m256i _mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i _mm256_maskz_abs_epi64( __mmask8 k, __m256i a);
VPABSQ __m128i _mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i _mm_maskz_abs_epi64( __mmask8 k, __m128i a);
PABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m128i _mm_abs_epi8 (__m128i a)
```

VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPABSD/Q, see Exceptions Type E4.

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 63 /r[1] PACKSSWB *mm1, mm2/m64*	A	V/V	MMX	Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation.
66 0F 63 /r PACKSSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xxm1* using signed saturation.
NP 0F 6B /r[1] PACKSSDW *mm1, mm2/m64*	A	V/V	MMX	Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation.
66 0F 6B /r PACKSSDW *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 4 packed signed doubleword integers from *xmm1* and from *xxm2/m128* into 8 packed signed word integers in *xxm1* using signed saturation.
VEX.128.66.0F.WIG 63 /r VPACKSSWB *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Converts 8 packed signed word integers from *xmm2* and from *xmm3/m128* into 16 packed signed byte integers in *xmm1* using signed saturation.
VEX.128.66.0F.WIG 6B /r VPACKSSDW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Converts 4 packed signed doubleword integers from *xmm2* and from *xmm3/m128* into 8 packed signed word integers in *xmm1* using signed saturation.
VEX.256.66.0F.WIG 63 /r VPACKSSWB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 16 packed signed word integers from *ymm2* and from *ymm3/m256* into 32 packed signed byte integers in *ymm1* using signed saturation.
VEX.256.66.0F.WIG 6B /r VPACKSSDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 8 packed signed doubleword integers from *ymm2* and from *ymm3/m256* into 16 packed signed word integers in *ymm1*using signed saturation.
EVEX.128.66.0F.WIG 63 /r VPACKSSWB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from *xmm2* and from *xmm3/m128* into packed signed byte integers in *xmm1* using signed saturation under writemask k1.
EVEX.256.66.0F.WIG 63 /r VPACKSSWB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from *ymm2* and from *ymm3/m256* into packed signed byte integers in *ymm1* using signed saturation under writemask k1.
EVEX.512.66.0F.WIG 63 /r VPACKSSWB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Converts packed signed word integers from *zmm2* and from *zmm3/m512* into packed signed byte integers in *zmm1* using signed saturation under writemask k1.
EVEX.128.66.0F.W0 6B /r VPACKSSDW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from *xmm2* and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.

EVEX.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from *ymm2* and from ymm3/m256/m32bcst into packed signed word integers in ymm1 using signed saturation under writemask k1.
EVEX.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW	Converts packed signed doubleword integers from zmm2 and from zmm3/m512/m32bcst into packed signed word integers in zmm1 using signed saturation under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

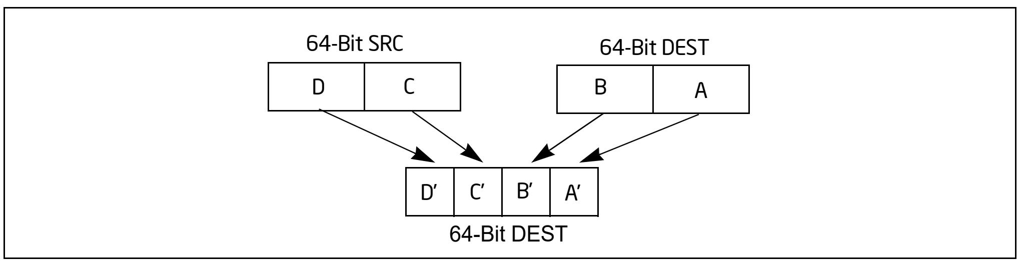

Figure 4-6. Operation of the PACKSSDW Instruction Using 64-bit Operands

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed word value is beyond the range of a signed byte value (i.e., greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second 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, updated conditional under the 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 first source operand is an XMM register. The second operand 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 destination register destination are unmodified.

Operation

PACKSSWB instruction (128-bit Legacy SSE version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)
```

PACKSSDW instruction (128-bit Legacy SSE version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);
DEST[MAXVL-1:128] (Unmodified)
```

VPACKSSWB instruction (VEX.128 encoded version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] ← 0;
```

VPACKSSDW instruction (VEX.128 encoded version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[MAXVL-1:128] ← 0;
```

VPACKSSWB instruction (VEX.256 encoded version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);
DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
```

```
    DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
    DEST[MAXVL-1:256] ← 0;
```

VPACKSSDW instruction (VEX.256 encoded version)

```
    DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);
    DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);
    DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);
    DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);
    DEST[MAXVL-1:256] ← 0;
```

VPACKSSWB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128]← SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
```

```
    TMP_DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] ← SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPACKSSDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;

TMP_DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] ← SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] ← SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] ← SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] ← SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] ← SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] ← SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] ← SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] ← SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] ← SaturateSignedDwordToSignedWord (SRC1[287:256]);
    TMP_DEST[287:272] ← SaturateSignedDwordToSignedWord (SRC1[319:288]);
    TMP_DEST[303:288] ← SaturateSignedDwordToSignedWord (SRC1[351:320]);
    TMP_DEST[319:304] ← SaturateSignedDwordToSignedWord (SRC1[383:352]);
    TMP_DEST[335:320] ← SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
    TMP_DEST[351:336] ← SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
    TMP_DEST[367:352] ← SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
    TMP_DEST[383:368] ← SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

    TMP_DEST[399:384] ← SaturateSignedDwordToSignedWord (SRC1[415:384]);
    TMP_DEST[415:400] ← SaturateSignedDwordToSignedWord (SRC1[447:416]);
    TMP_DEST[431:416] ← SaturateSignedDwordToSignedWord (SRC1[479:448]);
```

```
    TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
    TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
    TMP_DEST[479:464] ← SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
    TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
    TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32( __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi32( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi32( __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m128i _mm_mask_packs_epi32( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW__m128i _mm_maskz_packs_epi32( __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m512i _mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_maskz_packs_epi16( __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m256i _mm256_mask_packs_epi16( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m256i _mm256_maskz_packs_epi16( __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m128i _mm_mask_packs_epi16( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m128i _mm_maskz_packs_epi16( __mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.

PACKUSDW—Pack with Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW *xmm1, xmm2/m128*	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from *xmm1* and 4 packed signed doubleword integers from *xmm2/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.128.66.0F38 2B /r VPACKUSDW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Convert 4 packed signed doubleword integers from *xmm2* and 4 packed signed doubleword integers from *xmm3/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.256.66.0F38 2B /r VPACKUSDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Convert 8 packed signed doubleword integers from *ymm2* and 8 packed signed doubleword integers from *ymm3/m256* into 16 packed unsigned word integers in *ymm1* using unsigned saturation.
EVEX.128.66.0F38.W0 2B /r VPACKUSDW *xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst*	C	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.
EVEX.256.66.0F38.W0 2B /r VPACKUSDW *ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst*	C	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.
EVEX.512.66.0F38.W0 2B /r VPACKUSDW *zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst*	C	V/V	AVX512BW	Convert packed signed doubleword integers from *zmm2* and packed signed doubleword integers from *zmm3/m512/m32bcst* into packed unsigned word integers in *zmm1* using unsigned saturation under writemask k1.

Instruction Operand Encoding

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

Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second 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 register, updated conditionally under the 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 first source operand is an XMM register. The second operand 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 destination register destination are unmodified.

Operation

PACKUSDW (Legacy SSE instruction)

```
TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] ← (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] ← (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] ← (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] ← (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] ← (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] ← (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
DEST[MAXVL-1:128] (Unmodified)
```

PACKUSDW (VEX.128 encoded version)

```
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAXVL-1:128] ← 0;
```

VPACKUSDW (VEX.256 encoded version)

```
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
```

```
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
TMP[207:192] ← (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] ← (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
DEST[MAXVL-1:256] ← 0;
```

VPACKUSDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] ← (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] ← (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] ← (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] ← (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
IF VL >= 256
    TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
    DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
    TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
    DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
```

```
    TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
    DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
    TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
    DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
    TMP[207:192] ← (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
    DEST[207:192] ← (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
    TMP[223:208] ← (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
    DEST[223:208] ← (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
    TMP[239:224] ← (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
    DEST[239:224] ← (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
    TMP[255:240] ← (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
    DEST[255:240] ← (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
FI;
IF VL >= 512
    TMP[271:256] ← (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
    DEST[271:256] ← (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256] ;
    TMP[287:272] ← (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
    DEST[287:272] ← (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272] ;
    TMP[303:288] ← (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
    DEST[303:288] ← (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288] ;
    TMP[319:304] ← (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
    DEST[319:304] ← (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304] ;
    TMP[335:320] ← (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
    DEST[335:304] ← (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64] ;
    TMP[351:336] ← (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
    DEST[351:336] ← (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336] ;
    TMP[367:352] ← (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
    DEST[367:352] ← (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352] ;
    TMP[383:368] ← (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
    DEST[383:368] ← (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368] ;
    TMP[399:384] ← (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
    DEST[399:384] ← (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384] ;
    TMP[415:400] ← (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
    DEST[415:400] ← (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400] ;
    TMP[431:416] ← (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
    DEST[431:416] ← (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416] ;
    TMP[447:432] ← (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
    DEST[447:432] ← (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432] ;
    TMP[463:448] ← (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
    DEST[463:448] ← (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448] ;
    TMP[475:464] ← (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
    DEST[475:464] ← (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464] ;
    TMP[491:476] ← (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
    DEST[491:476] ← (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476] ;
    TMP[511:492] ← (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
    DEST[511:492] ← (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492] ;
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i _mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i _mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i _mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i _mm256_packus_epi32(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 67 /r[1] PACKUSWB *mm, mm/m64*	A	V/V	MMX	Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation.
66 0F 67 /r PACKUSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.128.66.0F.WIG 67 /r VPACKUSWB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Converts 8 signed word integers from *xmm2* and 8 signed word integers from *xmm3/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.256.66.0F.WIG 67 /r VPACKUSWB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 16 signed word integers from *ymm2* and 16signed word integers from *ymm3/m256* into 32 unsigned byte integers in *ymm1* using unsigned saturation.
EVEX.128.66.0F.WIG 67 /r VPACKUSWB *xmm1{k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Converts signed word integers from *xmm2* and signed word integers from *xmm3/m128* into unsigned byte integers in *xmm1* using unsigned saturation under writemask k1.
EVEX.256.66.0F.WIG 67 /r VPACKUSWB *ymm1{k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Converts signed word integers from *ymm2* and signed word integers from *ymm3/m256* into unsigned byte integers in *ymm1* using unsigned saturation under writemask k1.
EVEX.512.66.0F.WIG 67 /r VPACKUSWB *zmm1{k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Converts signed word integers from *zmm2* and signed word integers from *zmm3/m512* into unsigned byte integers in *zmm1* using unsigned saturation under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Converts 4, 8, 16 or 32 signed word integers from the destination operand (first operand) and 4, 8, 16 or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: 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 and EVEX.128 encoded versions: 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 register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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

PACKUSWB (with 64-bit operands)

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

PACKUSWB (Legacy SSE instruction)

```
DEST[7:0]←SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (VEX.128 encoded version)

```
DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

```
    DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[MAXVL-1:128] ← 0;
```

VPACKUSWB (VEX.256 encoded version)

```
    DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
    DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
    DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
    DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
    DEST[39:32] ←SaturateSignedWordToUnsignedByte (SRC1[79:64]);
    DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
    DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
    DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
    DEST[71:64] ←SaturateSignedWordToUnsignedByte (SRC2[15:0]);
    DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    DEST[143:136] ←SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    DEST[151:144] ←SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    DEST[159:152] ←SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
```

VPACKUSWB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

```
TMP_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);
```

```
    TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPACKUSWB__m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB__m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);
```

```
PACKUSWB:     __m64 _mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB:  __m128i _mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB:    __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);
```

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FC /r[1] PADDB *mm, mm/m64*	A	V/V	MMX	Add packed byte integers from *mm/m64* and *mm*.
NP 0F FD /r[1] PADDW *mm, mm/m64*	A	V/V	MMX	Add packed word integers from *mm/m64* and *mm*.
NP 0F FE /r[1] PADDD *mm, mm/m64*	A	V/V	MMX	Add packed doubleword integers from *mm/m64* and *mm*.
NP 0F D4 /r[1] PADDQ *mm, mm/m64*	A	V/V	MMX	Add packed quadword integers from *mm/m64* and *mm*.
66 0F FC /r PADDB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
66 0F D4 /r PADDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed quadword integers from *xmm2/m128* and *xmm1*.
VEX.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1*.
VEX.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed word integers from *xmm2, xmm3/m128* and *store in xmm1*.
VEX.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed quadword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1*.
EVEX.128.66.0F.WIG FC /r VPADDB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.128.66.0F.WIG FD /r VPADDW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed word integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.128.66.0F.W0 FE /r VPADDD *xmm1 {k1}{z}, xmm2,* xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from *xmm2, and xmm3/m128/m32bcst* and store in *xmm1* using writemask k1.
EVEX.128.66.0F.W1 D4 /r VPADDQ *xmm1 {k1}{z}, xmm2,* xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from *xmm2, and xmm3/m128/m64bcst* and store in *xmm1* using writemask k1.
EVEX.256.66.0F.WIG FC /r VPADDB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.
EVEX.256.66.0F.WIG FD /r VPADDW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed word integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F.W0 FE /r VPADDD *ymm1 {k1}{z}, ymm2,* ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from *ymm2, ymm3/m256/m32bcst* and store in *ymm1* using writemask k1.
EVEX.256.66.0F.W1 D4 /r VPADDQ *ymm1 {k1}{z}, ymm2,* ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from *ymm2, ymm3/m256/m64bcst* and store in *ymm1* using writemask k1.
EVEX.512.66.0F.WIG FC /r VPADDB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed byte integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.512.66.0F.WIG FD /r VPADDW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed word integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.512.66.0F.W0 FE /r VPADDD *zmm1 {k1}{z}, zmm2,* zmm3/m512/m32bcst	D	V/V	AVX512F	Add packed doubleword integers from *zmm2, zmm3/m512/m32bcst* and store in *zmm1* using writemask k1.
EVEX.512.66.0F.W1 D4 /r VPADDQ *zmm1 {k1}{z}, zmm2,* zmm3/m512/m64bcst	D	V/V	AVX512F	Add packed quadword integers from *zmm2, zmm3/m512/m64bcst* and store in *zmm1* using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

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

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

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 destination are cleared.

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 first source operand is an XMM register. The second operand 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

PADDB (with 64-bit operands)

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

PADDW (with 64-bit operands)

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

PADDD (with 64-bit operands)

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

PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE instruction)

```
DEST[7:0]← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← DEST[127:120] + SRC[127:120];
DEST[MAXVL-1:128] (Unmodified)
```

PADDW (Legacy SSE instruction)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← DEST[127:112] + SRC[127:112];
DEST[MAXVL-1:128] (Unmodified)
```

PADDD (Legacy SSE instruction)
```
DEST[31:0]← DEST[31:0]  + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96]← DEST[127:96] + SRC[127:96];
DEST[MAXVL-1:128] (Unmodified)
```

PADDQ (Legacy SSE instruction)
```
DEST[63:0]← DEST[63:0]  + SRC[63:0];
DEST[127:64]← DEST[127:64] + SRC[127:64];
DEST[MAXVL-1:128] (Unmodified)
```

VPADDB (VEX.128 encoded instruction)
```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← SRC1[127:120] + SRC2[127:120];
DEST[MAXVL-1:128] ← 0;
```

VPADDW (VEX.128 encoded instruction)
```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← SRC1[127:112] + SRC2[127:112];
DEST[MAXVL-1:128] ← 0;
```

VPADDD (VEX.128 encoded instruction)
```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← SRC1[127:96] + SRC2[127:96];
DEST[MAXVL-1:128] ← 0;
```

VPADDQ (VEX.128 encoded instruction)
```
DEST[63:0]← SRC1[63:0]  + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[MAXVL-1:128] ← 0;
```

VPADDB (VEX.256 encoded instruction)
```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248]← SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 encoded instruction)
```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)
DEST[255:240]← SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 encoded instruction)
```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)
DEST[255:224] ← SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 encoded instruction)
```
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];
```

VPADDB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDD (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*
        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*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDQ (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] + 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*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPADDB__m512i _mm512_add_epi8 ( __m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 ( __m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32( __m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32( __mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32( __mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32( __mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64( __m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64( __mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64( __mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64( __mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b );
PADDW __m128i _mm_add_epi16 ( __m128i a, __m128i b);
PADDD  __m128i _mm_add_epi32 ( __m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 ( __m128i a, __m128i b);
```

VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b);
VPADDW __m256i _mm256_add_epi16 (__m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 (__m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 (__m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_si64(__m64 m1, __m64 m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPADDD/Q, see Exceptions Type E4.

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EC /r[1] PADDSB *mm, mm/m64*	A	V/V	MMX	Add packed signed byte integers from *mm/m64 and mm* and saturate the results.
66 0F EC /r PADDSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
NP 0F ED /r[1] PADDSW *mm, mm/m64*	A	V/V	MMX	Add packed signed word integers from *mm/m64 and mm* and saturate the results.
66 0F ED /r PADDSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.128.66.0F.WIG EC /r VPADDSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed signed byte integers from *xmm3/m128* and *xmm2* saturate the results.
VEX.128.66.0F.WIG ED /r VPADDSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed signed word integers from *xmm3/m128* and *xmm2* and saturate the results.
VEX.256.66.0F.WIG EC /r VPADDSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.256.66.0F.WIG ED /r VPADDSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
EVEX.128.66.0F.WIG EC /r VPADDSB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed signed byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EC /r VPADDSB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EC /r VPADDSB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed signed byte integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG ED /r VPADDSW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed signed word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG ED /r VPADDSW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG ED /r VPADDSW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed signed word integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

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.

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 register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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

PADDSB (with 64-bit operands)

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

PADDSB (with 128-bit operands)

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

VPADDSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

VPADDSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PADDSW (with 64-bit operands)

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

PADDSW (with 128-bit operands)

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

VPADDSW (VEX.128 encoded version)

```
    DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] ← 0
```

VPADDSW (VEX.256 encoded version)

```
    DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB: __m64 _mm_adds_pi8(__m64 m1, __m64 m2)

(V)PADDSB: __m128i _mm_adds_epi8 (__m128i a, __m128i b)

VPADDSB: __m256i _mm256_adds_epi8 (__m256i a, __m256i b)

PADDSW: __m64 _mm_adds_pi16(__m64 m1, __m64 m2)

(V)PADDSW: __m128i _mm_adds_epi16 (__m128i a, __m128i b)

VPADDSW: __m256i _mm256_adds_epi16 (__m256i a, __m256i b)

VPADDSB__m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW__m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB__m512i _mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i _mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB__m512i _mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i _mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB__m256i _mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i _mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB__m256i _mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i _mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB__m128i _mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i _mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB__m128i _mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i _mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DC /r[1] PADDUSB *mm, mm/m64*	A	V/V	MMX	Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results.
66 0F DC /r PADDUSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
NP 0F DD /r[1] PADDUSW *mm, mm/m64*	A	V/V	MMX	Add packed unsigned word integers from *mm/m64 and mm* and saturate the results.
66 0F DD /r PADDUSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.
VEX.128.660F.WIG DC /r VPADDUSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed unsigned byte integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.128.66.0F.WIG DD /r VPADDUSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed unsigned word integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.256.66.0F.WIG DC /r VPADDUSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed unsigned byte integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.256.66.0F.WIG DD /r VPADDUSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed unsigned word integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
EVEX.128.66.0F.WIG DC /r VPADDUSB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DC /r VPADDUSB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DC /r VPADDUSB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed unsigned byte integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.
EVEX.128.66.0F.WIG DD /r VPADDUSW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DD /r VPADDUSW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.

EVEX.512.66.0F.WIG DD /r VPADDUSW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed unsigned word integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

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.

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 destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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

PADDUSB (with 64-bit operands)

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

PADDUSB (with 128-bit operands)

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

VPADDUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

VPADDUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (with 128-bit operands)

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

VPADDUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[MAXVL-1:128] ← 0
```

VPADDUSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDUSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDUSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB: __m64 _mm_adds_pu8(__m64 m1, __m64 m2)

PADDUSW: __m64 _mm_adds_pu16(__m64 m1, __m64 m2)

(V)PADDUSB: __m128i _mm_adds_epu8 (__m128i a, __m128i b)

(V)PADDUSW: __m128i _mm_adds_epu16 (__m128i a, __m128i b)

VPADDUSB: __m256i _mm256_adds_epu8 (__m256i a, __m256i b)

VPADDUSW: __m256i _mm256_adds_epu16 (__m256i a, __m256i b)

VPADDUSB__m512i _mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW__m512i _mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB__m512i _mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB__m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB__m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB__m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB__m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB__m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A 0F /r ib[1] PALIGNR *mm1, mm2/m64, imm8*	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *mm1.*
66 0F 3A 0F /r ib PALIGNR *xmm1, xmm2/m128, imm8*	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *xmm1.*
VEX.128.66.0F3A.WIG 0F /r ib VPALIGNR *xmm1, xmm2, xmm3/m128, imm8*	B	V/V	AVX	Concatenate *xmm2* and *xmm3/m128*, extract byte aligned result shifted to the right by constant value in *imm8* and result is stored in *xmm1*.
VEX.256.66.0F3A.WIG 0F /r ib VPALIGNR *ymm1, ymm2, ymm3/m256, imm8*	B	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and *ymm3/m256* into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in *imm8* from each intermediate result, and two 16-byte results are stored in *ymm1*.
EVEX.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	C	V/V	AVX512VL AVX512BW	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
EVEX.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	C	V/V	AVX512VL AVX512BW	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.512.66.0F3A.WIG 0F /r ib VPALIGNR zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	C	V/V	AVX512BW	Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

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

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: 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.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

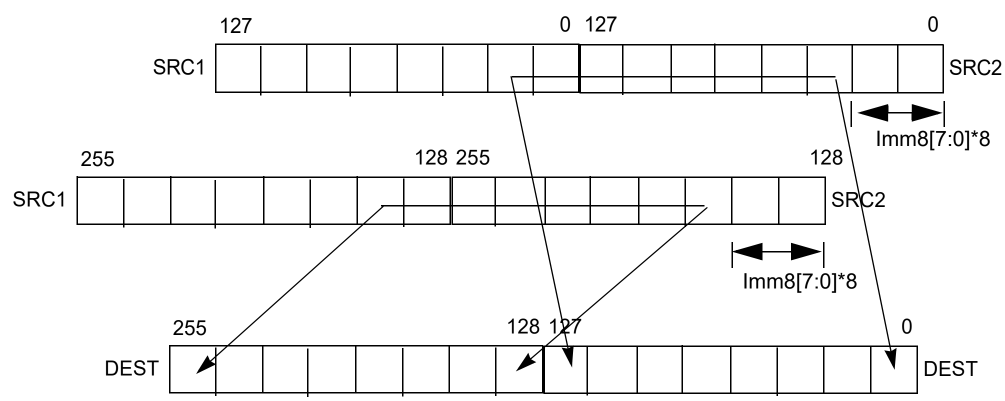

Figure 4-7. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

PALIGNR (with 128-bit operands)

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

VPALIGNR (VEX.128 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAXVL-1:128] ← 0
```

VPALIGNR (VEX.256 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAXVL-1:128] ← temp1[127:0]
```

VPALIGNR (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR l ← 0 TO VL-1 with increments of 128
    temp1[255:0] ← ((SRC1[l+127:l] << 128) OR SRC2[l+127:l])>>(imm8[7:0]*8);
    TMP_DEST[l+127:l] ← temp1[127:0]
ENDFOR;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
PALIGNR:      __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
(V)PALIGNR:   __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
VPALIGNR:     __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)
VPALIGNR __m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_maskz_alignr_epi8 ( __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DB /*r*[1] PAND *mm, mm/m64*	A	V/V	MMX	Bitwise AND *mm/m64* and *mm*.
66 0F DB /*r* PAND *xmm1*, *xmm2/m128*	A	V/V	SSE2	Bitwise AND of *xmm2/m128* and *xmm1*.
VEX.128.66.0F.WIG DB /r VPAND *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise AND of *xmm3/m128* and *xmm*.
VEX.256.66.0F.WIG DB /r VPAND *ymm1, ymm2, ymm3/.m256*	B	V/V	AVX2	Bitwise AND of *ymm2*, and *ymm3/m256* and store result in *ymm1*.
EVEX.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DB /*r* VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DB /*r* VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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.

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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: 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 versions: 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.

Operation

PAND (64-bit operand)

```
DEST ← DEST AND SRC
```

PAND (128-bit Legacy SSE version)

```
DEST ← DEST AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPAND (VEX.128 encoded version)

```
DEST ← SRC1 AND SRC2
DEST[MAXVL-1:128] ← 0
```

VPAND (VEX.256 encoded instruction)

```
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])
DEST[MAXVL-1:256] ← 0
```

VPANDD (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*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31: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
```

VPANDQ (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
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPANDD __m512i _mm512_and_epi32( __m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64( __m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_and_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_and_epi64( __mmask8 k, __m128i a, __m128i b);
PAND:    __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND:__m128i _mm_and_si128 ( __m128i a, __m128i b)
VPAND:       __m256i _mm256_and_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DF /*r*[1] PANDN *mm, mm/m64*	A	V/V	MMX	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /*r* PANDN *xmm1*, *xmm2/m128*	A	V/V	SSE2	Bitwise AND NOT of *xmm2/m128* and *xmm1*.
VEX.128.66.0F.WIG DF /r VPANDN *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise AND NOT of *xmm3/m128* and *xmm2*.
VEX.256.66.0F.WIG DF /r VPANDN *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Bitwise AND NOT of *ymm2*, and *ymm3/m256* and store result in *ymm1*.
EVEX.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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.

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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: 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 versions: 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.

Operation

PANDN (64-bit operand)

```
DEST ← NOT(DEST) AND SRC
```

PANDN (128-bit Legacy SSE version)

```
DEST ← NOT(DEST) AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPANDN (VEX.128 encoded version)

```
DEST ← NOT(SRC1) AND SRC2
DEST[MAXVL-1:128] ← 0
```

VPANDN (VEX.256 encoded instruction)

```
DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[MAXVL-1:256] ← 0
```

VPANDND (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*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[31:0])
            ELSE DEST[i+31:i] ← ((NOT SRC1[i+31:i]) 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
```

VPANDNQ (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] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] ← ((NOT SRC1[i+63:i]) 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
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPANDND __m512i _mm512_andnot_epi32( __m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_andnot_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m512i _mm512_andnot_epi64( __m512i a, __m512i b);
VPANDNQ __m512i _mm512_mask_andnot_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m512i _mm512_maskz_andnot_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m256i _mm256_mask_andnot_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_andnot_epi64( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_andnot_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_andnot_epi64( __mmask8 k, __m128i a, __m128i b);
PANDN: __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN:__m128i _mm_andnot_si128 ( __m128i a, __m128i b)
VPANDN:     __m256i _mm256_andnot_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PAUSE—Spin Loop Hint

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

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

```
Execute_Next_Instruction(DELAY);
```

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F E0 /*r*[1] PAVGB *mm1, mm2/m64*	A	V/V	SSE	Average packed unsigned byte integers from *mm2/m64* and *mm1* with rounding.
66 0F E0, /*r* PAVGB *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
NP 0F E3 /*r*[1] PAVGW *mm1, mm2/m64*	A	V/V	SSE	Average packed unsigned word integers from mm2/m64 and *mm1* with rounding.
66 0F E3 /*r* PAVGW *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.
VEX.128.66.0F.WIG E0 /r VPAVGB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Average packed unsigned byte integers from *xmm3/m128* and *xmm2* with rounding.
VEX.128.66.0F.WIG E3 /r VPAVGW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Average packed unsigned word integers from *xmm3/m128* and *xmm2* with rounding.
VEX.256.66.0F.WIG E0 /r VPAVGB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to *ymm1*.
VEX.256.66.0F.WIG E3 /r VPAVGW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Average packed unsigned word integers from *ymm2, ymm3/m256* with rounding to *ymm1*.
EVEX.128.66.0F.WIG E0 /r VPAVGB *xmm1* {k1}{z}, *xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from xmm2, and *xmm3/m128* with rounding and store to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E0 /r VPAVGB *ymm1* {k1}{z}, *ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E0 /r VPAVGB *zmm1* {k1}{z}, *zmm2, zmm3/m512*	C	V/V	AVX512BW	Average packed unsigned byte integers from zmm2, and *zmm3/m512* with rounding and store to zmm1 under writemask k1.
EVEX.128.66.0F.WIG E3 /r VPAVGW *xmm1* {k1}{z}, *xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from xmm2, *xmm3/m128* with rounding to xmm1 under writemask k1.
EVEX.256.66.0F.WIG E3 /r VPAVGW *ymm1* {k1}{z}, *ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from ymm2, *ymm3/m256* with rounding to ymm1 under writemask k1.
EVEX.512.66.0F.WIG E3 /r VPAVGW *zmm1* {k1}{z}, *zmm2, zmm3/m512*	C	V/V	AVX512BW	Average packed unsigned word integers from zmm2, *zmm3/m512* with rounding to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand 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.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: 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.

VEX.128 and EVEX.128 encoded versions: 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 register destination are zeroed.

Operation

PAVGB (with 64-bit operands)

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

PAVGW (with 64-bit operands)

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

PAVGB (with 128-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (with 128-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
DEST[127:112] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 encoded version)

```
    DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
    (* Repeat operation performed for bytes 2 through 15 *)
    DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
    DEST[MAXVL-1:128] ← 0
```

VPAVGW (VEX.128 encoded version)

```
    DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
    (* Repeat operation performed for 16-bit words 2 through 7 *)
    DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
    DEST[MAXVL-1:128] ← 0
```

VPAVGB (VEX.256 encoded instruction)

```
    DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 31)
    DEST[255:248] ← (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 encoded instruction)

```
    DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 15)
    DEST[255:14]) ← (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
VPAVGB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPAVGW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
                                    ; (* Temp sum before shifting is 17 bits *)
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAVGB: __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAVGW: __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAVGB: __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAVGW: __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB: __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW: __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PBLENDVB — Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB *xmm1, xmm2/m128, <XMM0>*	RM	V/V	SSE4_1	Select byte values from *xmm1* and *xmm2/m128* from mask specified in the high bit of each byte in *XMM0* and store the values into *xmm1.*
VEX.128.66.0F3A.W0 4C /r /is4 VPBLENDVB *xmm1, xmm2, xmm3/m128, xmm4*	RVMR	V/V	AVX	Select byte values from *xmm2* and *xmm3/m128* using mask bits in the specified mask register, *xmm4*, and store the values into *xmm1.*
VEX.256.66.0F3A.W0 4C /r /is4 VPBLENDVB *ymm1, ymm2, ymm3/m256, ymm4*	RVMR	V/V	AVX2	Select byte values from *ymm2* and *ymm3/m256* from mask specified in the high bit of each byte in *ymm4* and store the values into *ymm1*.

Instruction Operand Encoding

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

Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand 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 PBLENDVB 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.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM 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.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

PBLENDVB (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC[7:0];
ELSE DEST[7:0] ← DEST[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC[15:8];
```

```
ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ←    DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ←    DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120])
DEST[MAXVL-1:128] (Unmodified)
```

VPBLENDVB (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC2[87:80]
```

```
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ←    SRC1[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ←    SRC1[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
DEST[MAXVL-1:128] ← 0
```

VPBLENDVB (VEX.256 encoded version)

```
MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ←SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ←SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ←SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ←SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
IF (MASK[135] == 1) THEN DEST[135:128] ← SRC2[135:128];
ELSE DEST[135:128] ← SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144]
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152]
```

```
ELSE DEST[159:152] ← SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] ← SRC2[167:160]
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] ← SRC2[183:176]
ELSE DEST[183:176] ← SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] ← SRC2[191:184]
ELSE DEST[191:184] ← SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] ← SRC2[199:192]
ELSE DEST[199:192] ← SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] ← SRC2[207:200]
ELSE DEST[207:200] ← SRC1[207:200]
IF (MASK[215] == 1) THEN DEST[215:208] ← SRC2[215:208]
ELSE DEST[215:208] ← SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] ← SRC2[223:216]
ELSE DEST[223:216] ← SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] ← SRC2[231:224]
ELSE DEST[231:224] ← SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] ← SRC2[239:232]
ELSE DEST[239:232] ← SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] ← SRC2[247:240]
ELSE DEST[247:240] ← SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] ← SRC2[255:248]
ELSE DEST[255:248] ← SRC1[255:248]
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDVB: __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);

VPBLENDVB: __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

PBLENDW — Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Select words from *xmm1* and *xmm2/m128* from mask specified in *imm8* and store the values into *xmm1*.
VEX.128.66.0F3A.WIG 0E /r ib VPBLENDW *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Select words from *xmm2* and *xmm3/m128* from mask specified in *imm8* and store the values into *xmm1*.
VEX.256.66.0F3A.WIG 0E /r ib VPBLENDW *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Select words from *ymm2* and *ymm3/m256* from mask specified in *imm8* and store the values into *ymm1*.

Instruction Operand Encoding

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

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is “1", then the word is copied, else the word element in the destination operand is unchanged.

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

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

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.

Operation

PBLENDW (128-bit Legacy SSE version)

```
IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC[127:112]
```

```
ELSE DEST[127:112] ← DEST[127:112]
```

VPBLENDW (VEX.128 encoded version)

```
IF (imm8[0] = 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
DEST[MAXVL-1:128] ← 0
```

VPBLENDW (VEX.256 encoded version)

```
IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
IF (imm8[0] == 1) THEN DEST[143:128] ← SRC2[143:128]
ELSE DEST[143:128] ← SRC1[143:128]
IF (imm8[1] == 1) THEN DEST[159:144] ← SRC2[159:144]
ELSE DEST[159:144] ← SRC1[159:144]
IF (imm8[2] == 1) THEN DEST[175:160] ← SRC2[175:160]
ELSE DEST[175:160] ← SRC1[175:160]
IF (imm8[3] == 1) THEN DEST[191:176] ← SRC2[191:176]
ELSE DEST[191:176] ← SRC1[191:176]
IF (imm8[4] == 1) THEN DEST[207:192] ← SRC2[207:192]
ELSE DEST[207:192] ← SRC1[207:192]
IF (imm8[5] == 1) THEN DEST[223:208] ← SRC2[223:208]
ELSE DEST[223:208] ← SRC1[223:208]
IF (imm8[6] == 1) THEN DEST[239:224] ← SRC2[239:224]
ELSE DEST[239:224] ← SRC1[239:224]
IF (imm8[7] == 1) THEN DEST[255:240] ← SRC2[255:240]
ELSE DEST[255:240] ← SRC1[255:240]
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW: __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);

VPBLENDW: __m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1 and AVX2 = 0.

PCLMULQDQ — Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ *xmm1, xmm2/m128, imm8*	RMI	V/V	PCLMUL-QDQ	Carry-less multiplication of one quadword of xmm1 by one quadword of *xmm2/m128*, stores the 128-bit result in *xmm1*. The immediate is used to determine which quadwords of *xmm1* and *xmm2/m128* should be used.
VEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	Both PCL-MULQDQ and AVX flags	Carry-less multiplication of one quadword of *xmm2* by one quadword of *xmm3/m128*, stores the 128-bit result in *xmm1*. The immediate is used to determine which quadwords of *xmm2* and *xmm3/m128* should be used.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-13, other bits of the immediate byte are ignored.

Table 4-13. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[1][63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

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

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for Imm8.

Table 4-14. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ *xmm1, xmm2*	**0000_0000B**
PCLMULHQLQDQ *xmm1, xmm2*	**0000_0001B**
PCLMULLQHQDQ *xmm1, xmm2*	**0001_0000B**
PCLMULHQHQDQ *xmm1, xmm2*	**0001_0001B**

Operation

PCLMULQDQ

```
IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
DEST[127] ← 0;
DEST[MAXVL-1:128] (Unmodified)
```

VPCLMULQDQ

```
IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
```

```
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
DEST[MAXVL-1:127] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ: __m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4, additionally

#UD If VEX.L = 1.

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 74 /*r*[1] PCMPEQB *mm, mm/m64*	A	V/V	MMX	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /*r* PCMPEQB *xmm1*, *xmm2/m128*	A	V/V	SSE2	Compare packed bytes in *xmm2/m128* and *xmm1* for equality.
NP 0F 75 /*r*[1] PCMPEQW *mm, mm/m64*	A	V/V	MMX	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /*r* PCMPEQW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Compare packed words in *xmm2/m128* and *xmm1* for equality.
NP 0F 76 /*r*[1] PCMPEQD *mm, mm/m64*	A	V/V	MMX	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /*r* PCMPEQD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Compare packed doublewords in *xmm2/m128* and *xmm1* for equality.
VEX.128.66.0F.WIG 74 /r VPCMPEQB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed bytes in *xmm3/m128* and *xmm2* for equality.
VEX.128.66.0F.WIG 75 /r VPCMPEQW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed words in *xmm3/m128* and *xmm2* for equality.
VEX.128.66.0F.WIG 76 /r VPCMPEQD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed doublewords in *xmm3/m128* and *xmm2* for equality.
VEX.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.256.66.0F.WIG 75 /r VPCMPEQW *ymm1, ymm2, ymm3 /m256*	B	V/V	AVX2	Compare packed words in *ymm3/m256* and *ymm2* for equality.
VEX.256.66.0F.WIG 76 /r VPCMPEQD *ymm1, ymm2, ymm3 /m256*	B	V/V	AVX2	Compare packed doublewords in *ymm3/m256* and *ymm2* for equality.
EVEX.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.
EVEX.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128	D	V/V	AVX512VL AVX512BW	Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	D	V/V	AVX512VL AVX512BW	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	D	V/V	AVX512VL AVX512BW	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	D	V/V	AVX512VL AVX512BW	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

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

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

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.

EVEX encoded VPCMPEQD: 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 a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: 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. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQB (with 64-bit operands)

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

COMPARE_BYTES_EQUAL (SRC1, SRC2)

```
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPEQB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0
```

VPCMPEQB (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPEQB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] == SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPEQW (with 64-bit operands)

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

PCMPEQW (with 128-bit operands)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0
```

VPCMPEQW (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPEQW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] == SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPEQD (with 64-bit operands)

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

PCMPEQD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0
```

VPCMPEQD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPEQD (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
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] = 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;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB: __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW: __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD: __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB: __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW: __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD: __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB: __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW: __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD: __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPEQD, see Exceptions Type E4.
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.

PCMPEQQ — Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed qwords in *xmm2/m128* and *xmm1* for equality.
VEX.128.66.0F38.WIG 29 /r VPCMPEQQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed quadwords in *xmm3/m128* and *xmm2* for equality.
VEX.256.66.0F38.WIG 29 /r VPCMPEQQ *ymm1, ymm2, ymm3 /m256*	B	V/V	AVX2	Compare packed quadwords in *ymm3/m256* and *ymm2* for equality.
EVEX.128.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare Equal between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

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

Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

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

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

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.

EVEX encoded VPCMPEQQ: 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 a mask register updated according to the writemask k2.

Operation

PCMPEQQ (with 128-bit operands)

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

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0; FI;
    IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ←0; FI;
```

VPCMPEQQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPEQQ (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPEQQ (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] = SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] = 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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm256_cmpeq_epi64_mask(__m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm_cmpeq_epi64_mask(__m128i a, __m128i b);
VPCMPEQQ __mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ: __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ: __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPEQQ, see Exceptions Type E4.

PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 */r imm8* PCMPESTRI *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A 61 /r ib VPCMPESTRI *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

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

Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to the count register (ECX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	ECX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r imm8 PCMPESTRM *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in *XMM0.*
VEX.128.66.0F3A 60 /r ib VPCMPESTRM *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in *XMM0.*

Instruction Operand Encoding

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

Description

The instruction compares data from two string fragments based on the encoded value in the imm8 contol byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Length1	Length2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /*r*[1] PCMPGTB *mm, mm/m64*	A	V/V	MMX	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /*r* PCMPGTB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 65 /*r*[1] PCMPGTW *mm, mm/m64*	A	V/V	MMX	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /*r* PCMPGTW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 66 /*r*[1] PCMPGTD *mm, mm/m64*	A	V/V	MMX	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /*r* PCMPGTD *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.
VEX.128.66.0F.WIG 64 /r VPCMPGTB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* for greater than.
VEX.128.66.0F.WIG 65 /r VPCMPGTW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed word integers in *xmm2* and *xmm3/m128* for greater than.
VEX.128.66.0F.WIG 66 /r VPCMPGTD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed doubleword integers in *xmm2* and *xmm3/m128* for greater than.
VEX.256.66.0F.WIG 64 /r VPCMPGTB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m256* for greater than.
VEX.256.66.0F.WIG 65 /r VPCMPGTW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed word integers in *ymm2* and *ymm3/m256* for greater than.
VEX.256.66.0F.WIG 66 /r VPCMPGTD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed doubleword integers in *ymm2* and *ymm3/m256* for greater than.
EVEX.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

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

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

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.

EVEX encoded VPCMPGTD: 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 a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: 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. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (with 64-bit operands)

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

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPGTB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTB (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTW (with 64-bit operands)

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

PCMPGTW (with 128-bit operands)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTW (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
```

```
        ELSE    DEST[j] ← 0                 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTD (with 64-bit operands)

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

PCMPGTD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] > 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;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB:__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW:__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD:__m64 _mm_cmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB:__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
(V)PCMPGTW:__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)
(V)DCMPGTD:__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)
VPCMPGTB: __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
VPCMPGTW: __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
VPCMPGTD: __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPGTD, see Exceptions Type E4.
EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

PCMPGTQ — Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ *xmm1,xmm2/m128*	A	V/V	SSE4_2	Compare packed signed qwords in *xmm2/m128* and *xmm1* for greater than.
VEX.128.66.0F38.WIG 37 /r VPCMPGTQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed qwords in *xmm2* and *xmm3/m128* for greater than.
VEX.256.66.0F38.WIG 37 /r VPCMPGTQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed qwords in *ymm2* and *ymm3/m256* for greater than.
EVEX.128.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.256.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.512.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

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

Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

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

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

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.

EVEX encoded VPCMPGTD/Q: 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 a mask register updated according to the writemask k2.

Operation

COMPARE_QWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[63:0] > SRC2[63:0]
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0; FI;
    IF SRC1[127:64] > SRC2[127:64]
    THEN DEST[127:64] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ←0; FI;
```

VPCMPGTQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTQ (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTQ (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
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] > SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] > 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
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask( __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask( __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask( __m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPGTQ:     __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ:       __m256i _mm256_cmpgt_epi64( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTQ, see Exceptions Type E4.

PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 */r imm8* PCMPISTRI *xmm1, xmm2/m128, imm8*	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI *xmm1, xmm2/m128, imm8*	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

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

Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag – Set if any byte/word of xmm1 is null, reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```
int    _mm_cmpistri (__m128i a, __m128i b, const int mode);
```

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);

int _mm_cmpistrc (__m128i a, __m128i b, const int mode);

int _mm_cmpistro (__m128i a, __m128i b, const int mode);

int _mm_cmpistrs (__m128i a, __m128i b, const int mode);

int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 */r imm8* PCMPISTRM *xmm1, xmm2/m128, imm8*	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in *XMM0.*
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM *xmm1, xmm2/m128, imm8*	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in *XMM0.*

Instruction Operand Encoding

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

Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM") generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag – Set if any byte/word of xmm1 is null, reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);

int _mm_cmpistrc (__m128i a, __m128i b, const int mode);

int _mm_cmpistro (__m128i a, __m128i b, const int mode);

int _mm_cmpistrs (__m128i a, __m128i b, const int mode);

int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PDEP — Parallel Bits Deposit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F38.W0 F5 /r PDEP *r32a, r32b, r/m32*	RVM	V/V	BMI2	Parallel deposit of bits from *r32b* using mask in *r/m32*, result is written to *r32a*.
VEX.LZ.F2.0F38.W1 F5 /r PDEP *r64a, r64b, r/m64*	RVM	V/N.E.	BMI2	Parallel deposit of bits from *r64b* using mask in *r/m64*, result is written to *r64a*.

Instruction Operand Encoding

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

Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

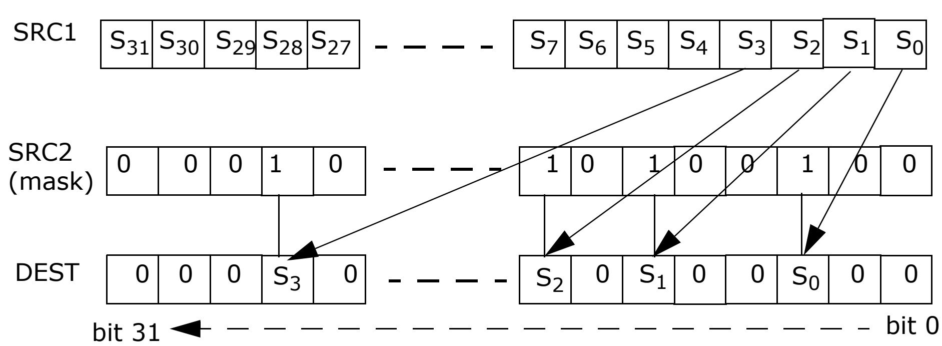

Figure 4-8. PDEP Example

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 ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

        IF MASK[ m] = 1 THEN
            DEST[ m] ← TEMP[ k];
            k ← k+ 1;
        FI
        m ← m+ 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PDEP: unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);

PDEP: unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 13.

PEXT — Parallel Bits Extract

Opcode/ Instruction	**Op/ En**	**64/32 -bit Mode**	**CPUID Feature Flag**	**Description**
VEX.LZ.F3.0F38.W0 F5 /r PEXT *r32a, r32b, r/m32*	RVM	V/V	BMI2	Parallel extract of bits from *r32b* using mask in *r/m32*, result is written to *r32a*.
VEX.LZ.F3.0F38.W1 F5 /r PEXT *r64a, r64b, r/m64*	RVM	V/N.E.	BMI2	Parallel extract of bits from *r64b* using mask in *r/m64*, result is written to *r64a*.

Instruction Operand Encoding

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

Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

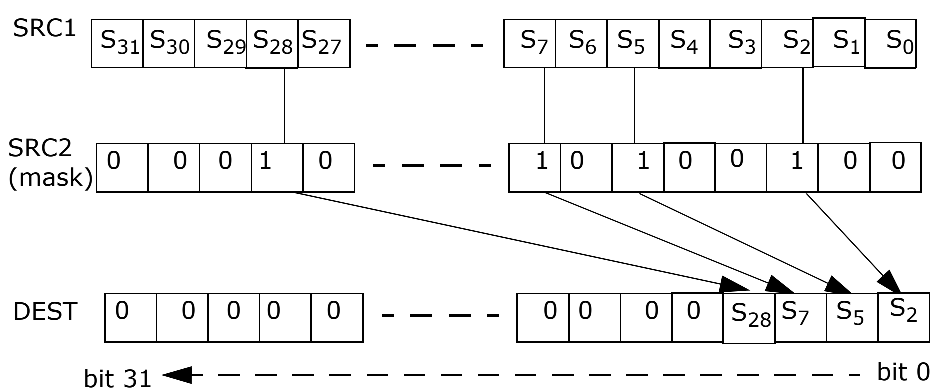

Figure 4-9. PEXT Example

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 ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

        IF MASK[ m] = 1 THEN
            DEST[ k] ← TEMP[ m];
            k ← k+ 1;
        FI
        m ← m+ 1;

OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PEXT: unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);

PEXT: unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 13.

PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB *reg/m8, xmm2, imm8*	A	V/V	SSE4_1	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *reg or m8.* The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD *r/m32, xmm2, imm8*	A	V/V	SSE4_1	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r/m32*.
66 REX.W 0F 3A 16 /r ib PEXTRQ *r/m64, xmm2, imm8*	A	V/N.E.	SSE4_1	Extract a qword integer value from *xmm2* at the source qword offset specified by *imm8* into *r/m64*.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB *reg/m8, xmm2, imm8*	A	V[1]/V	AVX	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *reg* or *m8*. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD *r32/m32, xmm2, imm8*	A	V/V	AVX	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r32/m32*.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ *r64/m64, xmm2, imm8*	A	V/I[2]	AVX	Extract a qword integer value from *xmm2* at the source dword offset specified by *imm8* into *r64/m64*.
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB reg/m8, xmm2, imm8	B	V/V	AVX512BW	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	B	V/V	AVX512DQ	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	B	V/N.E.[2]	AVX512DQ	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. 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	NA	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L"L must be

0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
CASE of
    PEXTRB: SEL ← COUNT[3:0];
            TEMP ← (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 ← TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] ← TEMP[7:0];
                    r64[63:8] ← ZERO_FILL; };
            ELSE
                    R32[7:0] ← TEMP[7:0];
                    r32[31:8] ← ZERO_FILL; };
            FI;
    PEXTRD:SEL ← COUNT[1:0];
            TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
            DEST ← TEMP;
    PEXTRQ: SEL ← COUNT[0];
            TEMP ← (Src >> SEL*64);
            DEST ← TEMP;
EASC:
```

VPEXTRTD/VPEXTRQ

```
IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset ← Imm8[0]
    r64/m64 ←(Src >> Src_Offset * 64)
ELSE
    Src_Offset ← Imm8[1:0]
    r32/m32 ← ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

VPEXTRB (dest=m8)

```
SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)
```

VPEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[3:0]
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] ←   ZERO_FILL;
ELSE
    SRC_Offset ←. Imm8[3:0];
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] ←   ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB: int _mm_extract_epi8 (__m128i src, const int ndx);

PEXTRD: int _mm_extract_epi32 (__m128i src, const int ndx);

PEXTRQ: __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /*r* ib[1] PEXTRW *reg, mm, imm8*	A	V/V	SSE	Extract the word specified by *imm8* from *mm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /*r* ib PEXTRW *reg, xmm, imm8*	A	V/V	SSE2	Extract the word specified by *imm8* from *xmm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW *reg/m16, xmm, imm8*	B	V/V	SSE4_1	Extract the word specified by *imm8* from *xmm* and copy it to lowest 16 bits of *reg or m16*. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW *reg, xmm1, imm8*	A	V[2]/V	AVX	Extract the word specified by *imm8* from *xmm1* and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW *reg/m16, xmm2, imm8*	B	V/V	AVX	Extract a word integer value from *xmm2* at the source word offset specified by *imm8* into *reg* or *m16*. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW reg, xmm1, imm8	A	V/V	AVX512BW	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW reg/m16, xmm2, imm8	C	V/V	AVX512BW	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
C	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PEXTRW instruction with 64-bit source operand)
          { SEL ← COUNT[1:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r64[15:0] ← TEMP[15:0];
            r64[63:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          { SEL ← COUNT[2:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r64[15:0] ← TEMP[15:0];
            r64[63:16] ← ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
          { SEL ← COUNT[1:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r32[15:0] ← TEMP[15:0];
            r32[31:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          { SEL ← COUNT[2:0];
            TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
            r32[15:0] ← TEMP[15:0];
            r32[31:16] ← ZERO_FILL; };
    FI;
FI;
```

VPEXTRW (dest=m16)

```
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)
```

```
VPEXTRW ( dest=reg)
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] ←   ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] ←   ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW:	int _mm_extract_pi16 (__m64 a, int n)
PEXTRW:	int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L’L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PHADDW/PHADDD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 01 /r[1] PHADDW *mm1, mm2/m64*	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to *mm1*.
66 0F 38 01 /r PHADDW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to *xmm1*.
NP 0F 38 02 /r PHADDD *mm1, mm2/m64*	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to *mm1*.
66 0F 38 02 /r PHADDD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.128.66.0F38.WIG 01 /r VPHADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to *xmm1*.
VEX.128.66.0F38.WIG 02 /r VPHADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.256.66.0F38.WIG 01 /r VPHADDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to *ymm1*.
VEX.256.66.0F38.WIG 02 /r VPHADDD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

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

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

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

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

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

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

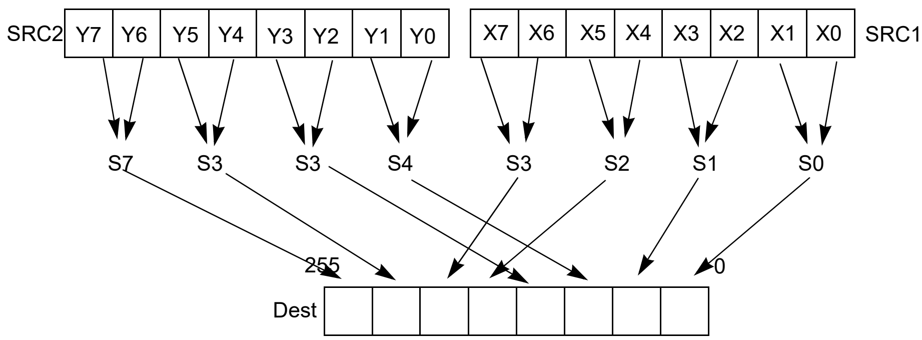

Figure 4-10. 256-bit VPHADDD Instruction Operation

Operation

PHADDW (with 64-bit operands)

```
mm1[15-0]  = mm1[31-16] + mm1[15-0];
mm1[31-16] = mm1[63-48] + mm1[47-32];
mm1[47-32] = mm2/m64[31-16] + mm2/m64[15-0];
mm1[63-48] = mm2/m64[63-48] + mm2/m64[47-32];
```

PHADDW (with 128-bit operands)

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

VPHADDW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[MAXVL-1:128] ← 0
```

VPHADDW (VEX.256 encoded version)
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[143:128] ← SRC1[159:144] + SRC1[143:128]
DEST[159:144] ← SRC1[191:176] + SRC1[175:160]
DEST[175:160] ← SRC1[223:208] + SRC1[207:192]
DEST[191:176] ← SRC1[255:240] + SRC1[239:224]
DEST[207:192] ← SRC2[127:112] + SRC2[143:128]
DEST[223:208] ← SRC2[159:144] + SRC2[175:160]
DEST[239:224] ← SRC2[191:176] + SRC2[207:192]
DEST[255:240] ← SRC2[223:208] + SRC2[239:224]

PHADDD (with 64-bit operands)
mm1[31-0] = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];

PHADDD (with 128-bit operands)
xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-96] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];

VPHADDD (VEX.128 encoded version)
DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[MAXVL-1:128] ← 0

VPHADDD (VEX.256 encoded version)
DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[159-128] ← SRC1[191-160] + SRC1[159-128]
DEST[191-160] ← SRC1[255-224] + SRC1[223-192]
DEST[223-192] ← SRC2[191-160] + SRC2[159-128]
DEST[255-224] ← SRC2[255-224] + SRC2[223-192]

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW:	__m64 _mm_hadd_pi16 (__m64 a, __m64 b)
PHADDD:	__m64 _mm_hadd_pi32 (__m64 a, __m64 b)
(V)PHADDW:	__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
(V)PHADDD:	__m128i _mm_hadd_epi32 (__m128i a, __m128i b)
VPHADDW:	__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
VPHADDD:	__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHADDSW — Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 03 /r[1] PHADDSW *mm1, mm2/m64*	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to *mm1*.
66 0F 38 03 /r PHADDSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to *xmm1*.
VEX.128.66.0F38.WIG 03 /r VPHADDSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to *xmm1*.
VEX.256.66.0F38.WIG 03 /r VPHADDSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

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

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHADDSW (with 64-bit operands)

```
mm1[15-0]  = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

PHADDSW (with 128-bit operands)

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

VPHADDSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[MAXVL-1:128] ← 0
```

VPHADDSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128]= SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord( SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191-160] + SRC2[159-128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW:	__m64 _mm_hadds_pi16 (__m64 a, __m64 b)
(V)PHADDSW:	__m128i _mm_hadds_epi16 (__m128i a, __m128i b)
VPHADDSW:	__m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHMINPOSUW — Packed Horizontal Word Minimum

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of *xmm1* and its index in the second-lowest word of *xmm1*.
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW *xmm1, xmm2/m128*	RM	V/V	AVX	Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of xmm1 and its index in the second-lowest word of *xmm1*.

Instruction Operand Encoding

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

Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

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

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination XMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHMINPOSUW (128-bit Legacy SSE version)

```
INDEX ← 0;
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN)
    THEN INDEX ← 1;  MIN ← SRC[31:16]; FI;
IF (SRC[47:32] < MIN)
    THEN INDEX ← 2;  MIN ← SRC[47:32]; FI;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
    THEN INDEX ← 7;  MIN ← SRC[127:112]; FI;
DEST[15:0] ← MIN;
DEST[18:16] ← INDEX;
DEST[127:19] ← 0000000000000000000000000000H;
```

VPHMINPOSUW (VEX.128 encoded version)

```
INDEX ← 0
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX ← 1; MIN ← SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX ← 2; MIN ← SRC[47:32]
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN) THEN INDEX ← 7; MIN ← SRC[127:112]
DEST[15:0] ← MIN
DEST[18:16] ← INDEX
DEST[127:19] ← 0000000000000000000000000000H
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW: __m128i _mm_minpos_epu16(__m128i packed_words);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 05 /r[1] PHSUBW *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to *mm1*.
66 0F 38 05 /r PHSUBW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to *xmm1*.
NP 0F 38 06 /r PHSUBD *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to *mm1*.
66 0F 38 06 /r PHSUBD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to *xmm1*.
VEX.128.66.0F38.WIG 05 /r VPHSUBW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to *xmm1*.
VEX.128.66.0F38.WIG 06 /r VPHSUBD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to *xmm1*.
VEX.256.66.0F38.WIG 05 /r VPHSUBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to *ymm1*.
VEX.256.66.0F38.WIG 06 /r VPHSUBD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to *ymm1*.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

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

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBW (with 64-bit operands)

```
mm1[15-0] = mm1[15-0] - mm1[31-16];
mm1[31-16] = mm1[47-32] - mm1[63-48];
mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];
```

PHSUBW (with 128-bit operands)

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

VPHSUBW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[MAXVL-1:128] ← 0
```

VPHSUBW (VEX.256 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]
```

PHSUBD (with 64-bit operands)

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

PHSUBD (with 128-bit operands)

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

VPHSUBD (VEX.128 encoded version)

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

VPHSUBD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW:	__m64 _mm_hsub_pi16 (__m64 a, __m64 b)
PHSUBD:	__m64 _mm_hsub_pi32 (__m64 a, __m64 b)
(V)PHSUBW:	__m128i _mm_hsub_epi16 (__m128i a, __m128i b)
(V)PHSUBD:	__m128i _mm_hsub_epi32 (__m128i a, __m128i b)
VPHSUBW:	__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)
VPHSUBD:	__m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHSUBSW — Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 07 /r[1] PHSUBSW *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to *mm1.*
66 0F 38 07 /r PHSUBSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to *xmm1.*
VEX.128.66.0F38.WIG 07 /r VPHSUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to *xmm1*.
VEX.256.66.0F38.WIG 07 /r VPHSUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to *ymm1*.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

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

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

PHSUBSW (with 128-bit operands)

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

VPHSUBSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[MAXVL-1:128] ← 0
```

VPHSUBSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128]= SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW:	__m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
(V)PHSUBSW:	__m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
VPHSUBSW:	__m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB *xmm1, r32/m8, imm8*	A	V/V	SSE4_1	Insert a byte integer value from *r32/m8* into *xmm1* at the destination element in *xmm1* specified by *imm8.*
66 0F 3A 22 /r ib PINSRD *xmm1, r/m32, imm8*	A	V/V	SSE4_1	Insert a dword integer value from *r/m32* into the *xmm1* at the destination element specified by *imm8.*
66 REX.W 0F 3A 22 /r ib PINSRQ *xmm1, r/m64, imm8*	A	V/N. E.	SSE4_1	Insert a qword integer value from *r/m64* into the *xmm1* at the destination element specified by *imm8.*
VEX.128.66.0F3A.W0 20 /r ib VPINSRB *xmm1, xmm2, r32/m8, imm8*	B	V[1]/V	AVX	Merge a byte integer value from *r32/m8* and rest from *xmm2* into *xmm1* at the byte offset in *imm8*.
VEX.128.66.0F3A.W0 22 /r ib VPINSRD *xmm1, xmm2, r/m32, imm8*	B	V/V	AVX	Insert a dword integer value from *r32/m32* and rest from *xmm2* into *xmm1* at the dword offset in *imm8*.
VEX.128.66.0F3A.W1 22 /r ib VPINSRQ *xmm1, xmm2, r/m64, imm8*	B	V/I[2]	AVX	Insert a qword integer value from *r64/m64* and rest from *xmm2* into *xmm1* at the qword offset in *imm8*.
EVEX.128.66.0F3A.WIG 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	C	V/V	AVX512BW	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
EVEX.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	C	V/V	AVX512DQ	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
EVEX.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	C	V/N.E.[2]	AVX512DQ	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. 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	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	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

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

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 register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

```
CASE OF
    PINSRB:  SEL ← COUNT[3:0];
             MASK ← (0FFH << (SEL * 8));
             TEMP ← (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL ← COUNT[1:0];
             MASK ← (0FFFFFFFFH << (SEL * 32));
             TEMP ← (((SRC << (SEL *32)) AND MASK) ;
    PINSRQ:  SEL ← COUNT[0]
             MASK ← (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP ← (((SRC << (SEL *64)) AND MASK) ;
ESAC;
        DEST ← ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX/EVEX encoded version)

```
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

VPINSRD (VEX/EVEX encoded version)

```
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

VPINSRQ (VEX/EVEX encoded version)

```
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRB:	__m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD:	__m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ:	__m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C4 /*r* ib[1] PINSRW *mm, r32/m16, imm8*	A	V/V	SSE	Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8.*
66 0F C4 /*r* ib PINSRW *xmm, r32/m16, imm8*	A	V/V	SSE2	Move the low word of *r32* or from *m16* into *xmm* at the word position specified by *imm8*.
VEX.128.66.0F.W0 C4 /r ib VPINSRW *xmm1, xmm2, r32/m16, imm8*	B	V[2]/V	AVX	Insert a word integer value from *r32/m16* and rest from *xmm2* into *xmm1* at the word offset in *imm8*.
EVEX.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	C	V/V	AVX512BW	Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	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

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

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. VEX.L must be 0, otherwise the instruction will #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L’L must be 0, otherwise the instruction will #UD.

Operation

PINSRW (with 64-bit source operand)

```
SEL ← COUNT AND 3H;
    CASE (Determine word position) OF
        SEL ← 0:     MASK ← 000000000000FFFFH;
        SEL ← 1:     MASK ← 00000000FFFF0000H;
        SEL ← 2:     MASK ← 0000FFFF00000000H;
        SEL ← 3:     MASK ← FFFF000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL ∗ 16)) AND MASK);
```

PINSRW (with 128-bit source operand)

```
SEL ← COUNT AND 7H;
    CASE (Determine word position) OF
        SEL ← 0:     MASK ← 0000000000000000000000000000FFFFH;
        SEL ← 1:     MASK ← 000000000000000000000000FFFF0000H;
        SEL ← 2:     MASK ← 00000000000000000000FFFF00000000H;
        SEL ← 3:     MASK ← 0000000000000000FFFF000000000000H;
        SEL ← 4:     MASK ← 000000000000FFFF0000000000000000H;
        SEL ← 5:     MASK ← 00000000FFFF00000000000000000000H;
        SEL ← 6:     MASK ← 0000FFFF000000000000000000000000H;
        SEL ← 7:     MASK ← FFFF0000000000000000000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL ∗ 16)) AND MASK);
```

VPINSRW (VEX/EVEX encoded version)

```
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

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

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

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L’L > 0.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 38 04 /r[1] PMADDUBSW *mm1, mm2/m64*	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *mm1*.
66 0F 38 04 /r PMADDUBSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.128.66.0F38.WIG 04 /r VPMADDUBSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.256.66.0F38.WIG 04 /r VPMADDUBSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *ymm1*.
EVEX.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (with 64 bit operands)

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

PMADDUBSW (with 128 bit operands)

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

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] ← 0
```

VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] ← 0
```

VPMADDUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW __m512i _mm512_maddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_maddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_maddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_maddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_maddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_maddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW: __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW: __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW: __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD *mm, mm/m64*	A	V/V	MMX	Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*.
66 0F F5 /r PMADDWD *xmm1, xmm2/m128*	A	V/V	SSE2	Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.128.66.0F.WIG F5 /r VPMADDWD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed word integers in *xmm2* by the packed word integers in *xmm3/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.256.66.0F.WIG F5 /r VPMADDWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed word integers in *ymm2* by the packed word integers in *ymm3/m256*, add adjacent doubleword results, and store in *ymm1*.
EVEX.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

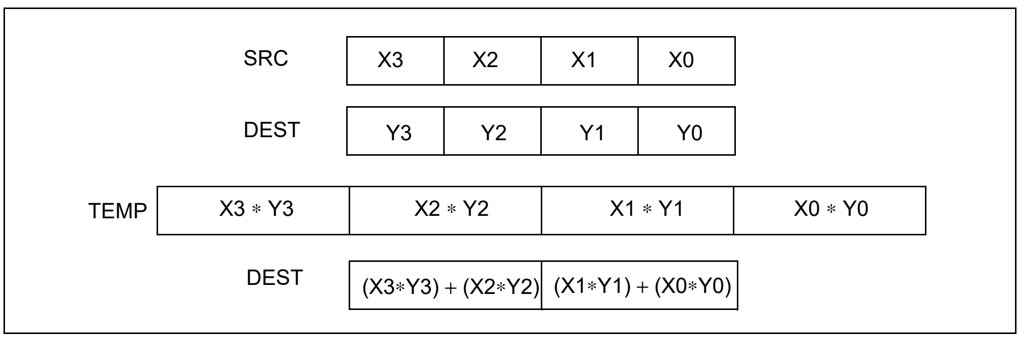

Figure 4-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (with 64-bit operands)

```
DEST[31:0] ← (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
DEST[63:32] ← (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
```

PMADDWD (with 128-bit operands)

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

VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[MAXVL-1:128] ← 0
```

VPMADDWD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[MAXVL-1:256] ← 0
```

VPMADDWD (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*
        THEN DEST[i+31:i] ← (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMADDWD __m512i _mm512_madd_epi16( __m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_madd_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_madd_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_mask_madd_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_madd_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm_mask_madd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i _mm_maskz_madd_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDWD:__m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD:__m128i _mm_madd_epi16 ( __m128i a, __m128i b)
VPMADDWD:__m256i _mm256_madd_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EE /*r*[1] PMAXSW *mm1, mm2/m64*	A	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXSW (64-bit operands)

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

PMAXSB (128-bit Legacy SSE version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] ←DEST[7:0];
    ELSE
        DEST[7:0] ←SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] ←DEST[127:120];
    ELSE
        DEST[127:120] ←SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSB (VEX.128 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ←SRC1[7:0];
    ELSE
        DEST[7:0] ←SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] ←SRC1[127:120];
    ELSE
        DEST[127:120] ←SRC2[127:120]; FI;
DEST[MAXVL-1:128] ←0
```

VPMAXSB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ←SRC1[7:0];
    ELSE
        DEST[7:0] ←SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ←SRC1[255:248];
    ELSE
        DEST[255:248] ←SRC2[255:248]; FI;
DEST[MAXVL-1:256] ←0
```

VPMAXSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMAXSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ←DEST[15:0];
    ELSE
        DEST[15:0] ←SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ←DEST[127:112];
    ELSE
        DEST[127:112] ←SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ←SRC1[15:0];
    ELSE
        DEST[15:0] ←SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ←SRC1[127:112];
    ELSE
        DEST[127:112] ←SRC2[127:112]; FI;
DEST[MAXVL-1:128] ←0
```

VPMAXSW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ←SRC1[15:0];
    ELSE
        DEST[15:0] ←SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] ←SRC1[255:240];
    ELSE
        DEST[255:240] ←SRC2[255:240]; FI;
DEST[MAXVL-1:256] ←0
```

VPMAXSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMAXSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ←DEST[31:0];
    ELSE
        DEST[31:0] ←SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] ←DEST[127:96];
    ELSE
        DEST[127:96] ←SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ←SRC1[31:0];
    ELSE
        DEST[31:0] ←SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] ←SRC1[127:96];
    ELSE
        DEST[127:96] ←SRC2[127:96]; FI;
DEST[MAXVL-1:128] ←0
```

VPMAXSD (VEX.256 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ←SRC1[31:0];
    ELSE
        DEST[31:0] ←SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] ←SRC1[255:224];
    ELSE
        DEST[255:224] ←SRC2[255:224]; FI;
DEST[MAXVL-1:256] ←0
```

VPMAXSD (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*THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMAXSQ (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
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8(__m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16(__m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8(__mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32(__m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64(__m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)
(V)PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 (__m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 (__m256i a, __m256i b)
VPMAXSD __m256i _mm256_max_epi32 (__m256i a, __m256i b);
PMAXSW:__m64 _mm_max_pi16(__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPMAXSD/Q, see Exceptions Type E4.
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb.

PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r[1] PMAXUB *mm1, mm2/m64*	A	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.*

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUB (64-bit operands)

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

PMAXUB (128-bit Legacy SSE version)

```
    IF DEST[7:0] >SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] >SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUB (VEX.128 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] >SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMAXUW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
          DEST[15:0] ← SRC1[15:0];
    ELSE
          DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
          DEST[255:240] ← SRC1[255:240];
    ELSE
          DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
          IF SRC1[i+15:i] > SRC2[i+15:i]
                THEN DEST[i+15:i] ← SRC1[i+15:i];
                ELSE DEST[i+15:i] ← SRC2[i+15:i];
          FI;
          ELSE
                IF *merging-masking*                    ; merging-masking
                      THEN *DEST[i+15:i] remains unchanged*
                      ELSE                              ; zeroing-masking
                            DEST[i+15:i] ← 0
                FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMAXUB __m512i _mm512_max_epu8( __m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8( __mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16( __m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16( __mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8( __mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_maskz_max_epu16( __mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8( __mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_maskz_max_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_epu8 ( __m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16 ( __m128i a, __m128i b)
VPMAXUB __m256i _mm256_max_epu8 ( __m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16 ( __m256i a, __m256i b);
PMAXUB: __m64 _mm_max_pu8(__m64 a, __m64 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] ← DEST[127:96];
    ELSE
        DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] ← SRC1[127:96];
    ELSE
        DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUD (VEX.256 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] ← SRC1[255:224];
    ELSE
        DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMAXUD (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* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPMAXUQ (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
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r[1] PMINSW *mm1, mm2/m64*	A	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSW (64-bit operands)

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

PMINSB (128-bit Legacy SSE version)

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSB (VEX.128 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINSB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMINSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINSW (VEX.256 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
         DEST[15:0] ← SRC1[15:0];
    ELSE
         DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
         DEST[255:240] ← SRC1[255:240];
    ELSE
         DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
         IF SRC1[i+15:i] < SRC2[i+15:i]
              THEN DEST[i+15:i] ← SRC1[i+15:i];
              ELSE DEST[i+15:i] ← SRC2[i+15:i];
         FI;
         ELSE
              IF *merging-masking*                  ; merging-masking
                   THEN *DEST[i+15:i] remains unchanged*
                   ELSE                              ; zeroing-masking
                        DEST[i+15:i] ← 0
              FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMINSB __m512i _mm512_min_epi8( __m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i _mm512_maskz_min_epi8( __mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16( __m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16( __mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i _mm256_maskz_min_epi8( __mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_maskz_min_epi16( __mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8( __mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16( __mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i _mm_min_epi8 ( __m128i a, __m128i b);
(V)PMINSW __m128i _mm_min_epi16 ( __m128i a, __m128i b)
VPMINSB __m256i _mm256_min_epi8 ( __m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16 ( __m256i a, __m256i b)
PMINSW:__m64 _mm_min_pi16 (__m64 a, __m64 b)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

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

PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
        DEST[127:96] ← DEST[127:96];
    ELSE
        DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSD (VEX.128 encoded version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
        DEST[127:96] ← SRC1[127:96];
    ELSE
        DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINSD (VEX.256 encoded version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] ← SRC1[255:224];
    ELSE
        DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINSD (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* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        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
```

VPMINSQ (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
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSD __m512i _mm512_min_epi32(__m512i a, __m512i b);
VPMINSD __m512i _mm512_mask_min_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSD __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSD __m256i _mm256_mask_min_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSD __m256i _mm256_maskz_min_epi32(__mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_mask_min_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b);
VPMINSD __m128i _mm_mask_min_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSD __m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_mask_min_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);
VPMINSD __m256i _mm256_min_epi32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r[1] PMINUB *mm1, mm2/m64*	A	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUB (for 64-bit operands)

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

PMINUB instruction for 128-bit operands:

```
    IF DEST[7:0] < SRC[7:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[15:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF DEST[127:120] < SRC[127:120] THEN
        DEST[127:120] ← DEST[127:120];
    ELSE
        DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUB (VEX.128 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[7:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 15th bytes in source and destination operands *)
    IF SRC1[127:120] < SRC2[127:120] THEN
        DEST[127:120] ← SRC1[127:120];
    ELSE
        DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINUB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMINUW instruction for 128-bit operands:

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINUW (VEX.256 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUB __m512i _mm512_min_epu8(__m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8(__mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16(__m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16(__mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_maskz_min_epu8(__mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu16(__mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i _mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i _mm_maskz_min_epu8(__mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_maskz_min_epu16(__mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 (__m128i a, __m128i b);
VPMINUB __m256i _mm256_min_epu8 (__m256i a, __m256i b)
VPMINUW __m256i _mm256_min_epu16 (__m256i a, __m256i b);
PMINUB: __m64 _m_min_pu8 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUD (128-bit Legacy SSE version)

```
PMINUD instruction for 128-bit operands:
    IF DEST[31:0] < SRC[31:0] THEN
         DEST[31:0] ← DEST[31:0];
    ELSE
         DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
         DEST[127:96] ← DEST[127:96];
    ELSE
         DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUD (VEX.128 encoded version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
         DEST[31:0] ← SRC1[31:0];
    ELSE
         DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
         DEST[127:96] ← SRC1[127:96];
    ELSE
         DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINUD (VEX.256 encoded version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
         DEST[31:0] ← SRC1[31:0];
    ELSE
         DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
         DEST[255:224] ← SRC1[255:224];
    ELSE
         DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINUD (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* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        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
```

VPMINUQ (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
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        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
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUD __m512i _mm512_min_epu32(__m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64(__m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32(__mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32 (__m128i a, __m128i b);
VPMINUD __m256i _mm256_min_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PMOVMSKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D7 /r[1] PMOVMSKB *reg, mm*	RM	V/V	SSE	Move a byte mask of *mm* to *reg*. The upper bits of r32 or r64 are zeroed
66 0F D7 /r PMOVMSKB *reg, xmm*	RM	V/V	SSE2	Move a byte mask of *xmm* to *reg*. The upper bits of r32 or r64 are zeroed
VEX.128.66.0F.WIG D7 /r VPMOVMSKB *reg, xmm1*	RM	V/V	AVX	Move a byte mask of *xmm1* to *reg*. The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMSKB *reg, ymm1*	RM	V/V	AVX2	Move a 32-bit mask of *ymm1* to *reg*. The upper bits of r64 are filled with zeros.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMSKB (with 64-bit source operand and r32)

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

(V)PMOVMSKB (with 128-bit source operand and r32)

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

VPMOVMSKB (with 256-bit source operand and r32)

r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 3rd through 31*)
r32[31] ← SRC[255];

PMOVMSKB (with 64-bit source operand and r64)

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] ← SRC[63];
r64[63:8] ← ZERO_FILL;
```

(V)PMOVMSKB (with 128-bit source operand and r64)

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] ← SRC[127];
r64[63:16] ← ZERO_FILL;
```

VPMOVMSKB (with 256-bit source operand and r64)

r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 31*)
r64[31] ← SRC[255];
r64[63:32] ← ZERO_FILL;

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB:	int _mm_movemask_pi8(__m64 a)
(V)PMOVMSKB:	int _mm_movemask_epi8 (__m128i a)
VPMOVMSKB:	int _mm256_movemask_epi8 (__m256i a)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

PMOVSX—Packed Move with Sign Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512BW	Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	AVX512VL AVX512F	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

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

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

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

Operation

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

Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)
```
DEST[31:0] ←SignExtend(SRC[7:0]);
DEST[63:32] ←SignExtend(SRC[15:8]);
DEST[95:64] ←SignExtend(SRC[23:16]);
DEST[127:96] ←SignExtend(SRC[31:24]);
```

Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)
```
DEST[63:0] ←SignExtend(SRC[7:0]);
DEST[127:64] ←SignExtend(SRC[15:8]);
```

Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)
```
DEST[31:0] ←SignExtend(SRC[15:0]);
DEST[63:32] ←SignExtend(SRC[31:16]);
DEST[95:64] ←SignExtend(SRC[47:32]);
DEST[127:96] ←SignExtend(SRC[63:48]);
```

Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)
```
DEST[63:0] ←SignExtend(SRC[15:0]);
DEST[127:64] ←SignExtend(SRC[31:16]);
```

Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)
```
DEST[63:0] ←SignExtend(SRC[31:0]);
DEST[127:64] ←SignExtend(SRC[63:32]);
```

VPMOVSXBW (EVEX encoded versions)
```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXBD (EVEX encoded versions)
```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXBQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXWD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXWQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXBW (VEX.256 encoded version)

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0
```

VPMOVSXBD (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVSXBQ (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] ← 0

VPMOVSXWD (VEX.256 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVSXWQ (VEX.256 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVSXDQ (VEX.256 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVSXBW (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_WORDDEST[127:0], SRC[127:0]()
DEST[MAXVL-1:128] ←0

VPMOVSXBD (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXBQ (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXWD (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXWQ (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXDQ (VEX.128 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVSXBW __m512i _mm512_cvtepi8_epi16(__m512i a);
VPMOVSXBW __m512i _mm512_mask_cvtepi8_epi16(__m512i a, __mmask32 k, __m512i b);
VPMOVSXBW __m512i _mm512_maskz_cvtepi8_epi16(__mmask32 k, __m512i b);
VPMOVSXBD __m512i _mm512_cvtepi8_epi32(__m512i a);
VPMOVSXBD __m512i _mm512_mask_cvtepi8_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXBD __m512i _mm512_maskz_cvtepi8_epi32(__mmask16 k, __m512i b);
VPMOVSXBQ __m512i _mm512_cvtepi8_epi64(__m512i a);
VPMOVSXBQ __m512i _mm512_mask_cvtepi8_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXBQ __m512i _mm512_maskz_cvtepi8_epi64(__mmask8 k, __m512i a);
VPMOVSXDQ __m512i _mm512_cvtepi32_epi64(__m512i a);
VPMOVSXDQ __m512i _mm512_mask_cvtepi32_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXDQ __m512i _mm512_maskz_cvtepi32_epi64(__mmask8 k, __m512i a);
VPMOVSXWD __m512i _mm512_cvtepi16_epi32(__m512i a);
VPMOVSXWD __m512i _mm512_mask_cvtepi16_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXWD __m512i _mm512_maskz_cvtepi16_epi32(__mmask16 k, __m512i a);
VPMOVSXWQ __m512i _mm512_cvtepi16_epi64(__m512i a);
VPMOVSXWQ __m512i _mm512_mask_cvtepi16_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXWQ __m512i _mm512_maskz_cvtepi16_epi64(__mmask8 k, __m512i a);
VPMOVSXBW __m256i _mm256_cvtepi8_epi16(__m256i a);
VPMOVSXBW __m256i _mm256_mask_cvtepi8_epi16(__m256i a, __mmask16 k, __m256i b);
VPMOVSXBW __m256i _mm256_maskz_cvtepi8_epi16(__mmask16 k, __m256i b);
VPMOVSXBD __m256i _mm256_cvtepi8_epi32(__m256i a);
VPMOVSXBD __m256i _mm256_mask_cvtepi8_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBD __m256i _mm256_maskz_cvtepi8_epi32(__mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_cvtepi8_epi64(__m256i a);
VPMOVSXBQ __m256i _mm256_mask_cvtepi8_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_maskz_cvtepi8_epi64(__mmask8 k, __m256i a);
VPMOVSXDQ __m256i _mm256_cvtepi32_epi64(__m256i a);
VPMOVSXDQ __m256i _mm256_mask_cvtepi32_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXDQ __m256i _mm256_maskz_cvtepi32_epi64(__mmask8 k, __m256i a);
VPMOVSXWD __m256i _mm256_cvtepi16_epi32(__m256i a);
VPMOVSXWD __m256i _mm256_mask_cvtepi16_epi32(__m256i a, __mmask16 k, __m256i b);
VPMOVSXWD __m256i _mm256_maskz_cvtepi16_epi32(__mmask16 k, __m256i a);

VPMOVSXWQ __m256i _mm256_cvtepi16_epi64(__m256i a);
VPMOVSXWQ __m256i _mm256_mask_cvtepi16_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXWQ __m256i _mm256_maskz_cvtepi16_epi64(__mmask8 k, __m256i a);
VPMOVSXBW __m128i _mm_mask_cvtepi8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBW __m128i _mm_maskz_cvtepi8_epi16(__mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_mask_cvtepi8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_maskz_cvtepi8_epi32(__mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_mask_cvtepi8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
VPMOVSXDQ __m128i _mm_mask_cvtepi32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXDQ __m128i _mm_maskz_cvtepi32_epi64(__mmask8 k, __m128i a);
VPMOVSXWD __m128i _mm_mask_cvtepi16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVSXWD __m128i _mm_maskz_cvtepi16_epi32(__mmask16 k, __m128i a);
VPMOVSXWQ __m128i _mm_mask_cvtepi16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXWQ __m128i _mm_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
PMOVSXBW __m128i _mm_ cvtepi8_epi16 (__m128i a);
PMOVSXBD __m128i _mm_ cvtepi8_epi32 (__m128i a);
PMOVSXBQ __m128i _mm_ cvtepi8_epi64 (__m128i a);
PMOVSXWD __m128i _mm_ cvtepi16_epi32 (__m128i a);
PMOVSXWQ __m128i _mm_ cvtepi16_epi64 (__m128i a);
PMOVSXDQ __m128i _mm_ cvtepi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

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

PMOVZX—Packed Move with Zero Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38 30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512BW	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	AVX512VL AVX512F	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Legacy, VEX and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

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

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

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask.The destination register is XMM, YMM or ZMM Register.

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

Operation

Packed_Zero_Extend_BYTE_to_WORD(DEST, SRC)

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

Packed_Zero_Extend_BYTE_to_DWORD(DEST, SRC)

```
DEST[31:0] ←ZeroExtend(SRC[7:0]);
DEST[63:32] ←ZeroExtend(SRC[15:8]);
DEST[95:64] ←ZeroExtend(SRC[23:16]);
DEST[127:96] ←ZeroExtend(SRC[31:24]);
```

Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)

```
DEST[63:0] ←ZeroExtend(SRC[7:0]);
DEST[127:64] ←ZeroExtend(SRC[15:8]);
```

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)

```
DEST[31:0] ←ZeroExtend(SRC[15:0]);
DEST[63:32] ←ZeroExtend(SRC[31:16]);
DEST[95:64] ←ZeroExtend(SRC[47:32]);
DEST[127:96] ←ZeroExtend(SRC[63:48]);
```

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)

```
DEST[63:0] ←ZeroExtend(SRC[15:0]);
DEST[127:64] ←ZeroExtend(SRC[31:16]);
```

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] ←ZeroExtend(SRC[31:0]);
DEST[127:64] ←ZeroExtend(SRC[63:32]);
```

VPMOVZXBW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXBD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXBQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXWD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
```

```
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXWQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXBW (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXBD (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVZXBQ (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] ← 0

VPMOVZXWD (VEX.256 encoded version)
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXWQ (VEX.256 encoded version)
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVZXDQ (VEX.256 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXBW (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] ←0

VPMOVZXBD (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXBQ (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXWD (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXWQ (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXDQ (VEX.128 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] ←0

PMOVZXBW

Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBD

Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBQ

Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWD

Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWQ

Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXDQ

Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVZXBW __m512i _mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m512i _mm512_mask_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBW __m512i _mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i b);
VPMOVZXBD __m512i _mm512_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m512i _mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXBD __m512i _mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i b);
VPMOVZXBQ __m512i _mm512_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m512i _mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m512i _mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m512i _mm512_cvtepu32_epi64(__m256i a);
VPMOVZXDQ __m512i _mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXDQ __m512i _mm512_maskz_cvtepu32_epi64(__mmask8 k, __m256i a);
VPMOVZXWD __m512i _mm512_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m512i _mm512_mask_cvtepu16_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXWD __m512i _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m512i _mm512_cvtepu16_epi64(__m256i a);
VPMOVZXWQ __m512i _mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXWQ __m512i _mm512_maskz_cvtepu16_epi64(__mmask8 k, __m256i a);
VPMOVZXBW __m256i _mm256_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m256i _mm256_mask_cvtepu8_epi16(__m256i a, __mmask16 k, __m128i b);
VPMOVZXBW __m256i _mm256_maskz_cvtepu8_epi16(__mmask16 k, __m128i b);
VPMOVZXBD __m256i _mm256_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m256i _mm256_mask_cvtepu8_epi32(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBD __m256i _mm256_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m256i _mm256_mask_cvtepu8_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m256i _mm256_cvtepu32_epi64(__m128i a);
VPMOVZXDQ __m256i _mm256_mask_cvtepu32_epi64(__m256i a, __mmask8 k, __m128i b);

VPMOVZXDQ __m256i _mm256_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m256i _mm256_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m256i _mm256_mask_cvtepu16_epi32(__m256i a, __mmask16 k, __m128i b);
VPMOVZXWD __m256i _mm256_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m256i _mm256_cvtepu16_epi64(__m128i a);
VPMOVZXWQ __m256i _mm256_mask_cvtepu16_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m256i _mm256_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
VPMOVZXBW __m128i _mm_mask_cvtepu8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i _mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_mask_cvtepu8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_mask_cvtepu8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m128i _mm_mask_cvtepu32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXDQ __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m128i _mm_mask_cvtepu16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVZXWD __m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i a);
VPMOVZXWQ __m128i _mm_mask_cvtepu16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
PMOVZXBW __m128i _mm_ cvtepu8_epi16 (__m128i a);
PMOVZXBD __m128i _mm_ cvtepu8_epi32 (__m128i a);
PMOVZXBQ __m128i _mm_ cvtepu8_epi64 (__m128i a);
PMOVZXWD __m128i _mm_ cvtepu16_epi32 (__m128i a);
PMOVZXWQ __m128i _mm_ cvtepu16_epi64 (__m128i a);
PMOVZXDQ __m128i _mm_ cvtepu32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

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

PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

Instruction Operand Encoding

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

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an 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 is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

VPMULDQ (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] ← SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[31:0])
                ELSE DEST[i+63:i] ← SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMULDQ (VEX.256 encoded version)

```
DEST[63:0] ←SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] ←SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[191:128] ←SignExtend64( SRC1[159:128]) * SignExtend64( SRC2[159:128])
DEST[255:192] ←SignExtend64( SRC1[223:192]) * SignExtend64( SRC2[223:192])
DEST[MAXVL-1:256] ←0
```

VPMULDQ (VEX.128 encoded version)

```
DEST[63:0] ←SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] ←SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[MAXVL-1:128] ←0
```

PMULDQ (128-bit Legacy SSE version)

```
DEST[63:0] ←SignExtend64( DEST[31:0]) * SignExtend64( SRC[31:0])
DEST[127:64] ←SignExtend64( DEST[95:64]) * SignExtend64( SRC[95:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULDQ __m512i _mm512_mul_epi32(__m512i a, __m512i b);
VPMULDQ __m512i _mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m512i _mm512_maskz_mul_epi32( __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m256i _mm256_mask_mul_epi32( __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULDQ __m128i _mm_mask_mul_epi32( __mmask8 k, __m128i a, __m128i b);
(V)PMULDQ __m128i _mm_mul_epi32( __m128i a, __m128i b);
VPMULDQ __m256i _mm256_mul_epi32( __m256i a, __m256i b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PMULHRSW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 0B /r[1] PMULHRSW *mm1, mm2/m64*	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *mm1*.
66 0F 38 0B /r PMULHRSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.128.66.0F38.WIG 0B /r VPMULHRSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.256.66.0F38.WIG 0B /r VPMULHRSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *ymm1*.
EVEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.
EVEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.
EVEX.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

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. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHRSW (with 64-bit operands)

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

PMULHRSW (with 128-bit operand)

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

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
```

```
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[MAXVL-1:128] ← 0
```

VPMULHRSW (VEX.256 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[143:128] ← temp8[16:1]
DEST[159:144] ← temp9[16:1]
DEST[175:160] ← temp10[16:1]
DEST[191:176] ← temp11[16:1]
DEST[207:192] ← temp12[16:1]
DEST[223:208] ← temp13[16:1]
DEST[239:224] ← temp14[16:1]
DEST[255:240] ← temp15[16:1]
DEST[MAXVL-1:256] ← 0
```

VPMULHRSW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
            DEST[i+15:i] ← tmp[16:1]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
```

```
            ELSE *zeroing-masking*            ; zeroing-masking
                DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16( __mmask8 k, __m128i a, __m128i b);
PMULHRSW:  __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW: __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW:__m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E4 /r[1] PMULHUW *mm1, mm2/m64*	A	V/V	SSE	Multiply the packed unsigned word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E4 /r PMULHUW *xmm1, xmm2/m128*	A	V/V	SSE2	Multiply the packed unsigned word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.128.66.0F.WIG E4 /r VPMULHUW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed unsigned word integers in *xmm2* and *xmm3/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.256.66.0F.WIG E4 /r VPMULHUW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed unsigned word integers in *ymm2* and *ymm3/m256*, and store the high 16 bits of the results in *ymm1*.
EVEX.128.66.0F.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

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. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

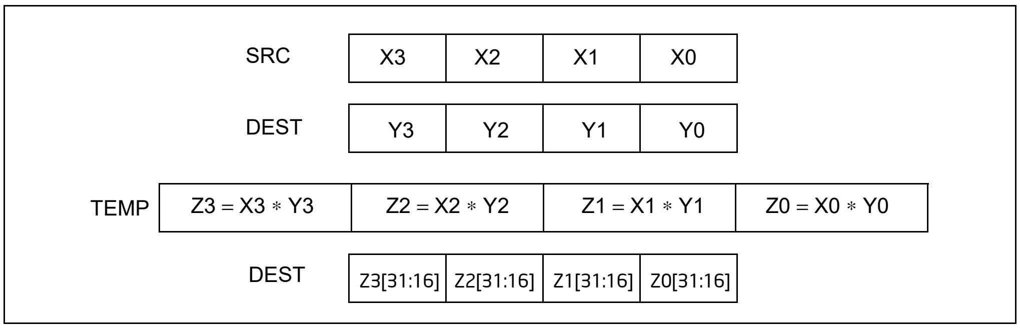

Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (with 64-bit operands)

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

PMULHUW (with 128-bit operands)

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

VPMULHUW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[MAXVL-1:128] ← 0
```

PMULHUW (VEX.256 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAXVL-1:256] ← 0
```

PMULHUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← tmp[31:16]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*           ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16( __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16( __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm_maskz_mulhi_epu16( __mmask8 k, __m128i a, __m128i b);
PMULHUW:__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW:__m128i _mm_mulhi_epu16 ( __m128i a, __m128i b)
VPMULHUW:__m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E5 /r[1] PMULHW *mm, mm/m64*	A	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E5 /r PMULHW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.128.66.0F.WIG E5 /r VPMULHW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed signed word integers in *xmm2* and *xmm3/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.256.66.0F.WIG E5 /r VPMULHW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed signed word integers in *ymm2* and *ymm3/m256*, and store the high 16 bits of the results in *ymm1*.
EVEX.128.66.0F.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

n 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

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. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHW (with 64-bit operands)

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

PMULHW (with 128-bit operands)

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

VPMULHW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[MAXVL-1:128] ← 0
```

PMULHW (VEX.256 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAXVL-1:256] ← 0
```

PMULHW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHW:__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW:__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW:__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

VPMULLQ (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 Temp[127:0] ← SRC1[i+63:i] * SRC2[63:0]
                   ELSE Temp[127:0] ← SRC1[i+63:i] * SRC2[i+63:i]
              FI;
              DEST[i+63:i] ← Temp[63:0]
         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
```

VPMULLD (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* THEN
              IF (EVEX.b = 1) AND (SRC2 *is memory*)
                   THEN Temp[63:0] ← SRC1[i+31:i] * SRC2[31:0]
                   ELSE Temp[63:0] ← SRC1[i+31:i] * SRC2[i+31:i]
              FI;
              DEST[i+31:i] ← Temp[31:0]
         ELSE
              IF *merging-masking*                ; merging-masking
                   *DEST[i+31:i] remains unchanged*
                   ELSE                           ; zeroing-masking
                        DEST[i+31:i] ← 0
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMULLD (VEX.256 encoded version)
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
Temp4[63:0] ← SRC1[159:128] * SRC2[159:128]
Temp5[63:0] ← SRC1[191:160] * SRC2[191:160]
Temp6[63:0] ← SRC1[223:192] * SRC2[223:192]
Temp7[63:0] ← SRC1[255:224] * SRC2[255:224]

DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[159:128] ← Temp4[31:0]
DEST[191:160] ← Temp5[31:0]
DEST[223:192] ← Temp6[31:0]
DEST[255:224] ← Temp7[31:0]
DEST[MAXVL-1:256] ← 0

VPMULLD (VEX.128 encoded version)
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[MAXVL-1:128] ← 0

PMULLD (128-bit Legacy SSE version)
Temp0[63:0] ← DEST[31:0] * SRC[31:0]
Temp1[63:0] ← DEST[63:32] * SRC[63:32]
Temp2[63:0] ← DEST[95:64] * SRC[95:64]
Temp3[63:0] ← DEST[127:96] * SRC[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m128i _mm_mask_mullo_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLD __m128i _mm_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i a, __m256i b);
PMULLD __m128i _mm_mullo_epi32(__m128i a, __m128i b);
VPMULLQ __m512i _mm512_mullo_epi64(__m512i a, __m512i b);
VPMULLQ __m512i _mm512_mask_mullo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);

VPMULLQ __m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64(__m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b);
VPMULLQ __m128i _mm_mullo_epi64(__m128i a, __m128i b);
VPMULLQ __m128i _mm_mask_mullo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLQ __m128i _mm_maskz_mullo_epi64(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D5 /*r*[1] PMULLW *mm, mm/m64*	A	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*.
66 0F D5 /*r* PMULLW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*.
VEX.128.66.0F.WIG D5 /r VPMULLW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed dword signed integers in *xmm2* and *xmm3/m128* and store the low 32 bits of each product in *xmm1*.
VEX.256.66.0F.WIG D5 /r VPMULLW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed signed word integers in *ymm2* and *ymm3/m256*, and store the low 16 bits of the results in *ymm1*.
EVEX.128.66.0F.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.256.66.0F.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.512.66.0F.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

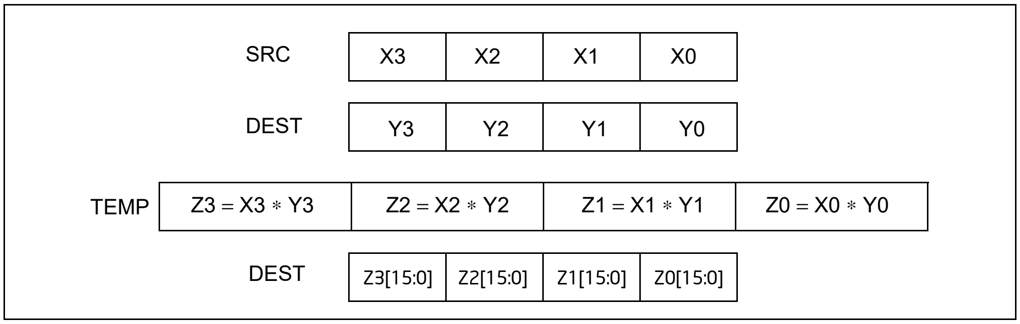

Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (with 64-bit operands)

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

PMULLW (with 128-bit operands)

```
    TEMP0[31:0] ←    DEST[15:0] ∗ SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] ←    DEST[31:16] ∗ SRC[31:16];
    TEMP2[31:0] ←    DEST[47:32] ∗ SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] ∗ SRC[63:48];
    TEMP4[31:0] ←    DEST[79:64] ∗ SRC[79:64];
    TEMP5[31:0] ←    DEST[95:80] ∗ SRC[95:80];
    TEMP6[31:0] ←    DEST[111:96] ∗ SRC[111:96];
    TEMP7[31:0] ←    DEST[127:112] ∗ SRC[127:112];
    DEST[15:0] ←     TEMP0[15:0];
    DEST[31:16] ←    TEMP1[15:0];
    DEST[47:32] ←    TEMP2[15:0];
    DEST[63:48] ←    TEMP3[15:0];
    DEST[79:64] ←    TEMP4[15:0];
    DEST[95:80] ←    TEMP5[15:0];
    DEST[111:96] ←   TEMP6[15:0];
    DEST[127:112] ← TEMP7[15:0];
DEST[MAXVL-1:256] ← 0
```

VPMULLW (VEX.128 encoded version)

```
Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[MAXVL-1:128] ← 0
```

PMULLW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← temp[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16( __mmask8 k, __m128i a, __m128i b);
PMULLW: __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW: __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW:__m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F4 /r[1] PMULUDQ *mm1*, *mm2/m64*	A	V/V	SSE2	Multiply unsigned doubleword integer in *mm1* by unsigned doubleword integer in *mm2/m64*, and store the quadword result in *mm1*.
66 0F F4 /r PMULUDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*.
VEX.128.66.0F.WIG F4 /r VPMULUDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply packed unsigned doubleword integers in *xmm2* by packed unsigned doubleword integers in *xmm3/m128*, and store the quadword results in *xmm1*.
VEX.256.66.0F.WIG F4 /r VPMULUDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply packed unsigned doubleword integers in *ymm2* by packed unsigned doubleword integers in *ymm3/m256*, and store the quadword results in *ymm1*.
EVEX.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an 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 is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

PMULUDQ (with 64-Bit operands)

```
    DEST[63:0] ← DEST[31:0] ∗ SRC[31:0];
```

PMULUDQ (with 128-Bit operands)

```
    DEST[63:0] ← DEST[31:0] ∗ SRC[31:0];
    DEST[127:64] ← DEST[95:64] ∗ SRC[95:64];
```

VPMULUDQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] ← 0
```

VPMULUDQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64
DEST[191:128] ← SRC1[159:128] * SRC2[159:128]
DEST[255:192] ← SRC1[223:192] * SRC2[223:192]
DEST[MAXVL-1:256] ← 0
```

VPMULUDQ (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] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[31:0] )
                ELSE DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i] )
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULUDQ __m512i _mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32(__mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32(__mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i _mm_maskz_mul_epu32(__mmask8 k, __m128i a, __m128i b);
PMULUDQ:__m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ:__m128i _mm_mul_epu32 (__m128i a, __m128i b)
VPMULUDQ:__m256i _mm256_mul_epu32(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

POP—Pop a Value from the Stack

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
O	opcode + rd (w)	NA	NA	NA
ZO	NA	NA	NA	NA

Description

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

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

The address size is used only when writing to a destination operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

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

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

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

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

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

Loading the SS register with a POP instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (POP ESP) before an event can be delivered. See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.* Intel recommends that software use the LSS instruction to load the SS register and ESP together.

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

Operation

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

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

FI;
```

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

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


PREOTECTED MODE OR COMPATIBILITY MODE;

IF SS is loaded;
    THEN
        IF segment selector is NULL
            THEN #GP(0);
        FI;
        IF segment selector index is outside descriptor table limits
            or segment selector's RPL ≠ CPL
```

```
                or segment is not a writable data segment
                or DPL ≠ CPL
                    THEN #GP(selector);
            FI;
            IF segment not marked present
                THEN #SS(selector);
                ELSE
                    SS ← segment selector;
                    SS ← segment descriptor;
            FI;
FI;

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

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

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with NULL segment selector.
	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.

POPA/POPAD—Pop All General-Purpose Registers

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

Instruction Operand Encoding

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

Description

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

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

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

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
ELSE
    IF OperandSize = 32 (* Instruction = POPAD *)
    THEN
        EDI ← Pop();
        ESI ← Pop();
        EBP ← Pop();
        Increment ESP by 4; (* Skip next 4 bytes of stack *)
        EBX ← Pop();
        EDX ← Pop();
        ECX ← Pop();
        EAX ← Pop();
    ELSE (* OperandSize = 16, instruction = POPA *)
        DI ← Pop();
        SI ← Pop();
        BP ← Pop();
        Increment ESP by 2; (* Skip next 2 bytes of stack *)
        BX ← Pop();
        DX ← Pop();
        CX ← Pop();
        AX ← Pop();
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

POPCNT — Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F B8 /r	POPCNT *r16, r/m16*	RM	Valid	Valid	POPCNT on *r/m16*
F3 0F B8 /r	POPCNT *r32, r/m32*	RM	Valid	Valid	POPCNT on *r/m32*
F3 REX.W 0F B8 /r	POPCNT *r64, r/m64*	RM	Valid	N.E.	POPCNT on *r/m64*

Instruction Operand Encoding

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

Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{       IF (SRC[ i] = 1) // i'th bit
        THEN Count++; FI;
}
DEST ← Count;
```

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

Intel C/C++ Compiler Intrinsic Equivalent

POPCNT: int _mm_popcnt_u32(unsigned int a);

POPCNT: int64_t _mm_popcnt_u64(unsigned __int64 a);

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 an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 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.POPCNT [Bit 23] = 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 an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 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.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.

POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9D	POPF	ZO	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	ZO	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPFQ	ZO	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

Instruction Operand Encoding

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

Description

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

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

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-15 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF[1], VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

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

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

Table 4-15. Effect of POPF/POPFD on the EFLAGS Register

Mode	Operand Size	CPL	IOPL	Flags																	Notes
				21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0	
				ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF	
Real-Address Mode (CR0.PE = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
Protected, Compatibility, and 64-Bit Modes (CR0.PE = 1 EFLAGS.VM = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	16	1-3	<CPL	N	N	N	N	N	0	S	N	S	S	N	S	S	S	S	S	S	
	16	1-3	≥CPL	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32, 64	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32, 64	1-3	<CPL	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S	
	32, 64	1-3	≥CPL	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
Virtual-8086 (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 0)	16	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
VME (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 1)	16	3	0-2	N/X	N/X	SV/X	N/X	N/X	0/X	S/X	N/X	S/X	S/X	N/X	S/X	S/X	S/X	S/X	S/X	S/X	2,3
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	

NOTES:

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

Key	
S	Updated from stack
SV	Updated from IF (bit 9) in FLAGS value on stack
N	No change in value
X	No EFLAGS update
0	Value is cleared

Operation

```
IF EFLAGS.VM = 0 (* Not in Virtual-8086 Mode *)
   THEN IF CPL = 0 OR CR0.PE = 0
       THEN
           IF OperandSize = 32;
               THEN
                   EFLAGS ← Pop(); (* 32-bit pop *)
                   (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                   VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
               ELSE IF (Operandsize = 64)
                   RFLAGS = Pop(); (* 64-bit pop *)
                   (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                   VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
```

```
                ELSE (* OperandSize = 16 *)
                    EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                    (* All non-reserved flags can be modified. *)
            FI;
        ELSE (* CPL > 0 *)
            IF OperandSize = 32
                THEN
                    IF CPL > IOPL
                        THEN
                            EFLAGS ← Pop(); (* 32-bit pop *)
                            (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                            IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                        ELSE
                            EFLAGS ← Pop(); (* 32-bit pop *)
                            (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                            IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    FI;
                ELSE IF (Operandsize = 64)
                    IF CPL > IOPL
                        THEN
                            RFLAGS ← Pop(); (* 64-bit pop *)
                            (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                            IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                        ELSE
                            RFLAGS ← Pop(); (* 64-bit pop *)
                            (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                            IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    FI;
                ELSE (* OperandSize = 16 *)
                    EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                    (* All non-reserved bits except IOPL can be modified; IOPL and all
                    reserved bits are unaffected. *)
            FI;
        FI;
    ELSE (* In virtual-8086 mode *)
        IF IOPL = 3
            THEN
                IF OperandSize = 32
                    THEN
                        EFLAGS ← Pop();
                        (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                        VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
                    ELSE
                        EFLAGS[15:0] ← Pop(); FI;
                        (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
                FI;
            ELSE (* IOPL < 3 *)
                IF (Operandsize = 32) OR (CR4.VME = 0)
                    THEN #GP(0); (* Trap to virtual-8086 monitor. *)
                    ELSE (* Operandsize = 16 and CR4.VME = 1 *)
                        tempFLAGS ← Pop();
                        IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
                            THEN #GP(0);
                            ELSE
```

```
                    EFLAGS.VIF ← tempFLAGS[9];
                    EFLAGS[15:0] ← tempFLAGS;
                    (* All non-reserved bits except IOPL and IF can be modified;
                    IOPL, IF, and all reserved bits are unaffected. *)
                FI;
            FI;
        FI;
FI;
```

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL < 3 and VME is not enabled.
	If IOPL < 3 and the 32-bit operand size is used.
	If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack.
	If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack.
	If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

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

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EB /r[1] POR *mm, mm/m64*	A	V/V	MMX	Bitwise OR of *mm/m64* and *mm*.
66 0F EB /*r* POR *xmm1*, *xmm2/m128*	A	V/V	SSE2	Bitwise OR of *xmm2/m128* and *xmm1*.
VEX.128.66.0F.WIG EB /r VPOR *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise OR of *xmm2/m128* and *xmm3*.
VEX.256.66.0F.WIG EB /r VPOR *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Bitwise OR of *ymm2/m256* and *ymm3*.
EVEX.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

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

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

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

EVEX encoded version: 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

Operation

POR (64-bit operand)

```
DEST ← DEST OR SRC
```

POR (128-bit Legacy SSE version)

```
DEST ← DEST OR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPOR (VEX.128 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[MAXVL-1:128] ← 0
```

VPOR (VEX.256 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[MAXVL-1:256] ← 0
```

VPORD (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* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
               ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                  ; merging-masking
               *DEST[i+31:i] remains unchanged*
               ELSE                               ; zeroing-masking
                  DEST[i+31:i] ← 0
            FI;
   FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32(__mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b,);
VPORD __m256i _mm256_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b);
VPORQ __m512i _mm512_or_epi64(__m512i a, __m512i b);
VPORQ __m512i _mm512_mask_or_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPORQ __m512i _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
VPORQ __m256i _mm256_or_epi64(__m256i a, int imm);
VPORQ __m256i _mm256_mask_or_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORQ __m256i _mm256_maskz_or_epi64(__mmask8 k, __m256i a, __m256i b);
VPORQ __m128i _mm_or_epi64(__m128i a, __m128i b);
VPORQ __m128i _mm_mask_or_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORQ __m128i _mm_maskz_or_epi64(__mmask8 k, __m128i a, __m128i b);
POR __m64 _mm_or_si64(__m64 m1, __m64 m2)
(V)POR: __m128i _mm_or_si128(__m128i m1, __m128i m2)
VPOR: __m256i _mm256_or_si256 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PREFETCH*h*—Prefetch Data Into Caches

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

Instruction Operand Encoding

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

Description

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

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
- T1 (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- T2 (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

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

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

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

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

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

Operation

```
FETCH (m8);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char *p, int i)

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

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /1 PREFETCHW m8	A	V/V	PREFETCHW	Move data from m8 closer to the processor in anticipation of a write.

Instruction Operand Encoding

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

Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates other cached copies in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHW instructions, or any other general instruction

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

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

Operation

```
FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);
```

Flags Affected

All flags are affected.

C/C++ Compiler Intrinsic Equivalent

```
void _m_prefetchw( 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

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r[1] PSADBW *mm1, mm2/m64*	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from *mm2 /m64* and *mm1*; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW *xmm1, xmm2/m128*	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from *xmm2 /m128* and *xmm1*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.128.66.0F.WIG F6 /r VPSADBW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from *xmm3 /m128* and *xmm2*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.256.66.0F.WIG F6 /r VPSADBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from *ymm3 /m256* and *ymm2*; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	C	V/V	AVX512BW	Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

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

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

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

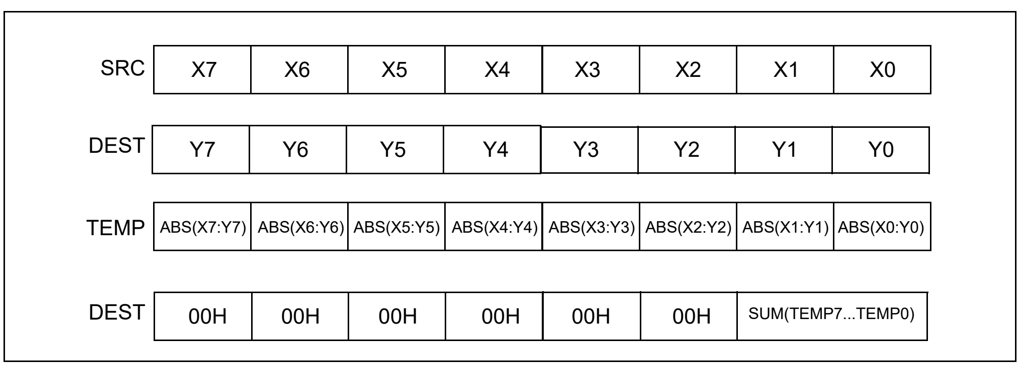

Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands

Operation

VPSADBW (EVEX encoded versions)

```
VL = 128, 256, 512
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H

IF VL >= 256
    (* Repeat operation for bytes 16 through 31*)
    TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
    DEST[143:128] ←SUM(TEMP16:TEMP23)
    DEST[191:144] ← 000000000000H
    DEST[207:192] ← SUM(TEMP24:TEMP31)
    DEST[223:208] ← 00000000000H
FI;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
    TEMP63 ← ABS(SRC1[511:504] - SRC2[511:504])
    DEST[271:256] ←SUM(TEMP0:TEMP7)
    DEST[319:272] ← 000000000000H
    DEST[335:320] ← SUM(TEMP8:TEMP15)
    DEST[383:336] ← 00000000000H
    DEST[399:384] ←SUM(TEMP16:TEMP23)
    DEST[447:400] ← 000000000000H
    DEST[463:448] ← SUM(TEMP24:TEMP31)
    DEST[511:464] ← 00000000000H
FI;
DEST[MAXVL-1:VL] ← 0
```

VPSADBW (VEX.256 encoded version)

```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[143:128] ←SUM(TEMP16:TEMP23)
DEST[191:144] ← 000000000000H
DEST[207:192] ← SUM(TEMP24:TEMP31)
DEST[223:208] ← 00000000000H
DEST[MAXVL-1:256] ← 0
```

VPSADBW (VEX.128 encoded version)
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[MAXVL-1:128] ← 0

PSADBW (128-bit Legacy SSE version)
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000
DEST[MAXVL-1:128] (Unmodified)

PSADBW (64-bit operand)
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 ← ABS(DEST[63:56] - SRC[63:56])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H

Intel C/C++ Compiler Intrinsic Equivalent

VPSADBW __m512i _mm512_sad_epu8(__m512i a, __m512i b)
PSADBW:__m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW:__m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW:__m256i _mm256_sad_epu8(__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 38 00 /r[1] PSHUFB *mm1, mm2/m64*	A	V/V	SSSE3	Shuffle bytes in *mm1* according to contents of *mm2/m64*.
66 0F 38 00 /r PSHUFB *xmm1, xmm2/m128*	A	V/V	SSSE3	Shuffle bytes in *xmm1* according to contents of *xmm2/m128*.
VEX.128.66.0F38.WIG 00 /r VPSHUFB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Shuffle bytes in *xmm2* according to contents of *xmm3/m128*.
VEX.256.66.0F38.WIG 00 /r VPSHUFB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Shuffle bytes in *ymm2* according to contents of *ymm3/m256*.
EVEX.128.66.0F38.WIG 00 /r VPSHUFB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1.
EVEX.256.66.0F38.WIG 00 /r VPSHUFB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1.
EVEX.512.66.0F38.WIG 00 /r VPSHUFB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

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

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask.

The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

Operation

PSHUFB (with 64 bit operands)

```
TEMP ← DEST
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif;
}
```

PSHUFB (with 128 bit operands)

```
TEMP ← DEST
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
     else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (VEX.128 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[MAXVL-1:128] ← 0
```

VPSHUFB (VEX.256 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7..(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
```

```
    endif
}
```

VPSHUFB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask ← (KL-1) & ~0xF                          // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1                              // dest
    IF kl[ i ] or no_masking
        index ← src.byte[ j ];
        IF index & 0x80
            Dest.byte[ j ] ← 0;
        ELSE
            index ← (index & 0xF) + (j & jmask);        // 16-element in-lane lookup
            Dest.byte[ j ] ← src.byte[ index ];
    ELSE if zeroing
        Dest.byte[ j ] ← 0;
DEST[MAXVL-1:VL] ← 0;
```

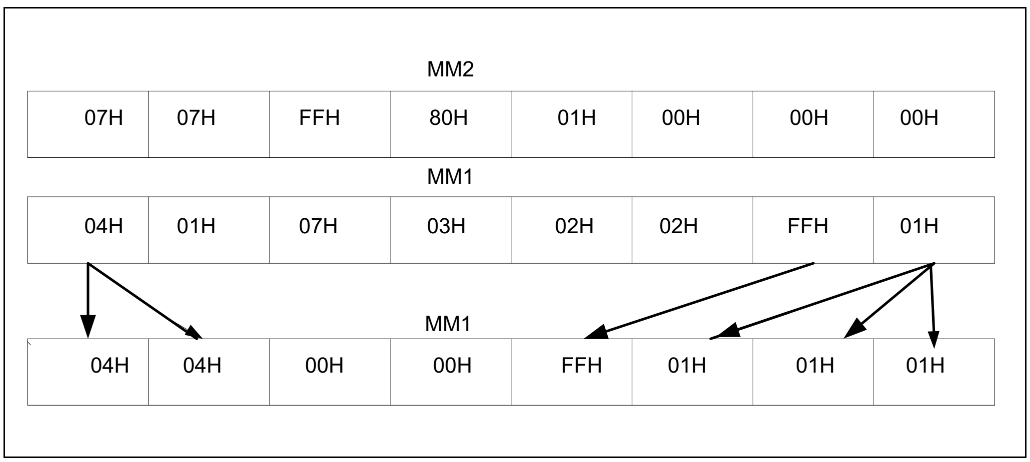

Figure 4-15. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD *xmm1*, *xmm2/m128, imm8*	A	V/V	SSE2	Shuffle the doublewords in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD *xmm1, xmm2/m128, imm8*	A	V/V	AVX	Shuffle the doublewords in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.256.66.0F.WIG 70 /r ib VPSHUFD *ymm1, ymm2/m256, imm8*	A	V/V	AVX2	Shuffle the doublewords in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.
EVEX.128.66.0F.W0 70 /r ib VPSHUFD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	B	V/V	AVX512VL AVX512F	Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 70 /r ib VPSHUFD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	B	V/V	AVX512VL AVX512F	Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 70 /r ib VPSHUFD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	B	V/V	AVX512F	Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

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

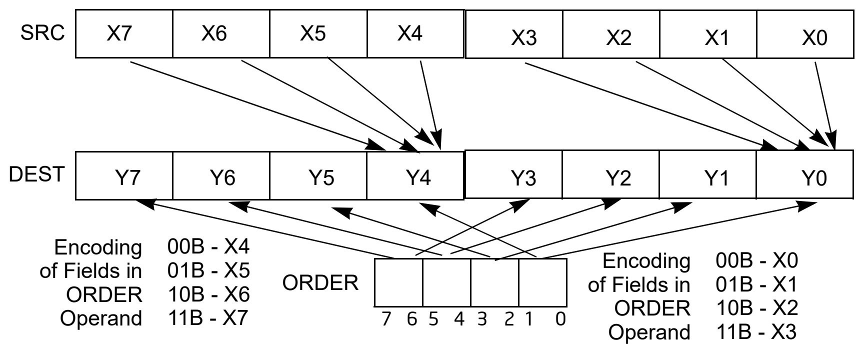

Figure 4-16. 256-bit VPSHUFD Instruction Operation

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

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

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

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an 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 updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

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

Operation

PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] ← 0
```

VPSHUFD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:256] ← 0
```

VPSHUFD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+31:i] ← SRC[31:0]
        ELSE TMP_SRC[i+31:i] ← SRC[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[63:32] ← (TMP_SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[95:64] ← (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[127:96] ← (TMP_SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 256
    TMP_DEST[159:128] ← (TMP_SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[191:160] ← (TMP_SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 512
    TMP_DEST[287:256] ← (TMP_SRC[383:256] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[319:288] ← (TMP_SRC[383:256] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[351:320] ← (TMP_SRC[383:256] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[383:352] ← (TMP_SRC[383:256] >> (ORDER[7:6] * 32))[31:0];
    TMP_DEST[415:384] ← (TMP_SRC[511:384] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[447:416] ← (TMP_SRC[511:384] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[479:448] ←(TMP_SRC[511:384] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[511:480] ← (TMP_SRC[511:384] >> (ORDER[7:6] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFD __m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, int n);
(V)PSHUFD:__m128i _mm_shuffle_epi32(__m128i a, int n)
VPSHUFD:__m256i _mm256_shuffle_epi32(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD	If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.

PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW *xmm1*, *xmm2/m128*, *imm8*	A	V/V	SSE2	Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW *xmm1*, *xmm2/m128*, *imm8*	A	V/V	AVX	Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW *ymm1*, *ymm2/m256*, *imm8*	A	V/V	AVX2	Shuffle the high words in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW	Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, 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 destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

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

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

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

Operation

PSHUFHW (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] ← 0
```

VPSHUFHW (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] ← SRC1[191:128]
DEST[207192] ← (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[MAXVL-1:256] ← 0
```

VPSHUFHW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
    TMP_DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
    TMP_DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
    TMP_DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[207:192] ← (SRC1 >> (imm[1:0] *16))[207:192]
    TMP_DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
    TMP_DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
    TMP_DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[335:320] ← (SRC1 >> (imm[1:0] *16))[335:320]
```

```
    TMP_DEST[351:336] ← (SRC1 >> (imm[3:2] * 16))[335:320]
    TMP_DEST[367:352] ← (SRC1 >> (imm[5:4] * 16))[335:320]
    TMP_DEST[383:368] ← (SRC1 >> (imm[7:6] * 16))[335:320]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[463:448] ← (SRC1 >> (imm[1:0] *16))[463:448]
    TMP_DEST[479:464] ← (SRC1 >> (imm[3:2] * 16))[463:448]
    TMP_DEST[495:480] ← (SRC1 >> (imm[5:4] * 16))[463:448]
    TMP_DEST[511:496] ← (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16( __mmask16 k, __m512i a, int n );
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFHW __m256i _mm256_maskz_shufflehi_epi16( __mmask8 k, __m256i a, int n );
VPSHUFHW __m128i _mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFHW __m128i _mm_maskz_shufflehi_epi16( __mmask8 k, __m128i a, int n );
(V)PSHUFHW:__m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW:__m256i _mm256_shufflehi_epi16(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

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

PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW *xmm1*, *xmm2/m128*, *imm8*	A	V/V	SSE2	Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW *xmm1, xmm2/m128, imm8*	A	V/V	AVX	Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW *ymm1, ymm2/m256, imm8*	A	V/V	AVX2	Shuffle the low words in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW	Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, 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 destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

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

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

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

Operation

PSHUFLW (128-bit Legacy SSE version)

```
DEST[15:0] ← (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFLW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[MAXVL-1:128] ← 0
```

VPSHUFLW (VEX.256 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC1[127:64]
DEST[143:128] ← (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] ← SRC1[255:192]
DEST[MAXVL-1:256] ← 0
```

VPSHUFLW (EVEX.U1.512 encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
    TMP_DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
    TMP_DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
    TMP_DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
    TMP_DEST[127:64] ← SRC1[127:64]
FI;
IF VL >= 256
    TMP_DEST[143:128] ← (SRC1 >> (imm[1:0] *16))[143:128]
    TMP_DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
    TMP_DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
    TMP_DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
    TMP_DEST[255:192] ← SRC1[255:192]
FI;
IF VL >= 512
    TMP_DEST[271:256] ← (SRC1 >> (imm[1:0] *16))[271:256]
    TMP_DEST[287:272] ← (SRC1 >> (imm[3:2] * 16))[271:256]
    TMP_DEST[303:288] ← (SRC1 >> (imm[5:4] * 16))[271:256]
    TMP_DEST[319:304] ← (SRC1 >> (imm[7:6] * 16))[271:256]
    TMP_DEST[383:320] ← SRC1[383:320]
```

```
    TMP_DEST[399:384] ← (SRC1 >> (imm[1:0] *16))[399:384]
    TMP_DEST[415:400] ← (SRC1 >> (imm[3:2] * 16))[399:384]
    TMP_DEST[431:416] ← (SRC1 >> (imm[5:4] * 16))[399:384]
    TMP_DEST[447:432] ← (SRC1 >> (imm[7:6] * 16))[399:384]
    TMP_DEST[511:448] ← SRC1[511:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n );
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16( __mmask16 k, __m512i a, int n );
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n );
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16( __mmask8 k, __m256i a, int n );
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n );
VPSHUFLW __m128i _mm_maskz_shufflelo_epi16( __mmask8 k, __m128i a, int n );
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

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

PSHUFW—Shuffle Packed Words

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

Instruction Operand Encoding

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

Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFW: __m64 _mm_shuffle_pi16(__m64 a, int n)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Table 22-7, “Exception Conditions for SIMD/MMX Instructions with Memory Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 08 /r[1] PSIGNB *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in *mm1* depending on the corresponding sign in *mm2/m64.*
66 0F 38 08 /r PSIGNB *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in *xmm1* depending on the corresponding sign in *xmm2/m128*.
NP 0F 38 09 /r[1] PSIGNW *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in *mm1* depending on the corresponding sign in *mm2/m128*.
66 0F 38 09 /r PSIGNW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in *xmm1* depending on the corresponding sign in *xmm2/m128*.
NP 0F 38 0A /r[1] PSIGND *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in *mm1* depending on the corresponding sign in *mm2/m128.*
66 0F 38 0A /r PSIGND *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in *xmm1* depending on the corresponding sign in *xmm2/m128.*
VEX.128.66.0F38.WIG 08 /r VPSIGNB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in *xmm2* depending on the corresponding sign in *xmm3/m128*.
VEX.128.66.0F38.WIG 09 /r VPSIGNW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed word integers in *xmm2* depending on the corresponding sign in *xmm3/m128*.
VEX.128.66.0F38.WIG 0A /r VPSIGND *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in *xmm2* depending on the corresponding sign in *xmm3/m128.*
VEX.256.66.0F38.WIG 08 /r VPSIGNB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed byte integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.
VEX.256.66.0F38.WIG 09 /r VPSIGNW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed 16-bit integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.
VEX.256.66.0F38.WIG 0A /r VPSIGND *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed doubleword integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

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

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

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Operation

PSIGNB (with 64 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes

IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]
```

PSIGNB (with 128 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]
```

VPSIGNB (VEX.128 encoded version)

```
DEST[127:0] ←BYTE_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

VPSIGNB (VEX.256 encoded version)

```
DEST[255:0] ←BYTE_SIGN_256b(SRC1, SRC2)
```

PSIGNW (with 64 bit operands)

```
    IF (SRC[15:0] < 0 )
        DEST[15:0] ← Neg(DEST[15:0])
    ELSEIF (SRC[15:0] = 0 )
        DEST[15:0] ← 0
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 3rd words
    IF (SRC[63:48] < 0 )
        DEST[63:48] ← Neg(DEST[63:48])
    ELSEIF (SRC[63:48] = 0 )
        DEST[63:48] ← 0
    ELSEIF (SRC[63:48] > 0 )
        DEST[63:48] ← DEST[63:48]
```

PSIGNW (with 128 bit operands)

```
    IF (SRC[15:0] < 0 )
        DEST[15:0] ← Neg(DEST[15:0])
    ELSEIF (SRC[15:0] = 0 )
        DEST[15:0] ← 0
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
    Repeat operation for 2nd through 7th words
    IF (SRC[127:112] < 0 )
        DEST[127:112] ← Neg(DEST[127:112])
    ELSEIF (SRC[127:112] = 0 )
        DEST[127:112] ← 0
    ELSEIF (SRC[127:112] > 0 )
        DEST[127:112] ← DEST[127:112]
```

VPSIGNW (VEX.128 encoded version)

```
DEST[127:0] ←WORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

VPSIGNW (VEX.256 encoded version)

```
DEST[255:0] ←WORD_SIGN(SRC1, SRC2)
```

PSIGND (with 64 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    IF (SRC[63:32] < 0 )
        DEST[63:32] ← Neg(DEST[63:32])
    ELSEIF (SRC[63:32] = 0 )
        DEST[63:32] ← 0
```

```
    ELSEIF (SRC[63:32] > 0 )
        DEST[63:32] ← DEST[63:32]
```

PSIGND (with 128 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    Repeat operation for 2nd through 3rd double words
    IF (SRC[127:96] < 0 )
        DEST[127:96] ← Neg(DEST[127:96])
    ELSEIF (SRC[127:96] = 0 )
        DEST[127:96] ← 0
    ELSEIF (SRC[127:96] > 0 )
        DEST[127:96] ← DEST[127:96]
```

VPSIGND (VEX.128 encoded version)

```
DEST[127:0] ←DWORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

VPSIGND (VEX.256 encoded version)

```
DEST[255:0] ←DWORD_SIGN(SRC1, SRC2)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB: __m64 _mm_sign_pi8 (__m64 a, __m64 b)

(V)PSIGNB: __m128i _mm_sign_epi8 (__m128i a, __m128i b)

VPSIGNB: __m256i _mm256_sign_epi8 (__m256i a, __m256i b)

PSIGNW: __m64 _mm_sign_pi16 (__m64 a, __m64 b)

(V)PSIGNW: __m128i _mm_sign_epi16 (__m128i a, __m128i b)

VPSIGNW: __m256i _mm256_sign_epi16 (__m256i a, __m256i b)

PSIGND: __m64 _mm_sign_pi32 (__m64 a, __m64 b)

(V)PSIGND: __m128i _mm_sign_epi32 (__m128i a, __m128i b)

VPSIGND: __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ *xmm1*, *imm8*	A	V/V	SSE2	Shift *xmm1* left by *imm8* bytes while shifting in 0s.
VEX.128.66.0F.WIG 73 /7 ib VPSLLDQ *xmm1, xmm2, imm8*	B	V/V	AVX	Shift *xmm2* left by *imm8* bytes while shifting in 0s and store result in *xmm1*.
VEX.256.66.0F.WIG 73 /7 ib VPSLLDQ *ymm1, ymm2, imm8*	B	V/V	AVX2	Shift *ymm2* left by *imm8* bytes while shifting in 0s and store result in *ymm1*.
EVEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1,xmm2/ m128, imm8	C	V/V	AVX512VL AVX512BW	Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8	C	V/V	AVX512VL AVX512BW	Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW	Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r, w)	imm8	NA	NA
B	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA

Description

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

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

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Operation

VPSLLDQ (EVEX.U1.512 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[255:128] ← SRC[255:128] << (TEMP * 8)
DEST[383:256] ← SRC[383:256] << (TEMP * 8)
DEST[511:384] ← SRC[511:384] << (TEMP * 8)
DEST[MAXVL-1:512] ← 0
```

VPSLLDQ (VEX.256 and EVEX.256 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[255:128] ← SRC[255:128] << (TEMP * 8)
DEST[MAXVL-1:256] ← 0
```

VPSLLDQ (VEX.128 and EVEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[MAXVL-1:128] ← 0
```

PSLLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSLLDQ:__m128i _mm_slli_si128 (__m128i a, int imm)
VPSLLDQ:__m256i _mm256_slli_si256 (__m256i a, const int imm)
VPSLLDQ __m512i _mm512_bslli_epi128 (__m512i a, const int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /r PSLLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 71 /6 ib PSLLW *mm1, imm8*	B	V/V	MMX	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib PSLLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F2 /r[1] PSLLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /r PSLLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib PSLLD *xmm1, imm8*	B	V/V	SSE2	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F3 /r[1] PSLLQ *mm, mm/m64*	A	V/V	MMX	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /r PSLLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ *mm, imm8*	B	V/V	MMX	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib PSLLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG F1 /r VPSLLW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.128.66.0F.WIG 71 /6 ib VPSLLW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* left by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG F2 /r VPSLLD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.128.66.0F.WIG 72 /6 ib VPSLLD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* left by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG F3 /r VPSLLQ *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift quadwords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.128.66.0F.WIG 73 /6 ib VPSLLQ *xmm1, xmm2, imm8*	D	V/V	AVX	Shift quadwords in *xmm2* left by *imm8* while shifting in 0s.
VEX.256.66.0F.WIG F1 /r VPSLLW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.256.66.0F.WIG 71 /6 ib VPSLLW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* left by *imm8* while shifting in 0s.

VEX.256.66.0F.WIG F2 /r VPSLLD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.256.66.0F.WIG 72 /6 ib VPSLLD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* left by *imm8* while shifting in 0s.
VEX.256.66.0F.WIG F3 /r VPSLLQ *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift quadwords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.256.66.0F.WIG 73 /6 ib VPSLLQ *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift quadwords in *ymm2* left by *imm8* while shifting in 0s.
EVEX.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

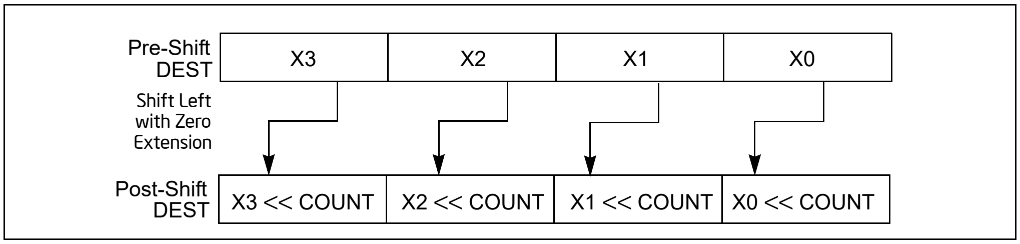

Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (with 64-bit operand)

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

PSLLD (with 64-bit operand)

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

PSLLQ (with 64-bit operand)

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

```
LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
```

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

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
```

```
    DEST[255:240] ←ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] ←ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLW (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)
```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)
```
(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) AND (SRC1 *is memory*)
                    THEN DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                    ELSE DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
                FI;
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*          ; zeroing-masking
                        DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLD (EVEX versions, xmm/m128)
```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*          ; zeroing-masking
                        DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding
```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLD (ymm, imm8) - VEX.256 encoding
```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLD (xmm, imm8) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLD (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)
```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLQ (EVEX versions, imm8)
```
(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 (SRC1 *is memory*)
                THEN DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VPSLLQ (EVEX versions, xmm/m128)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32( __mmask8 k, __m128i a, __m128i cnt);
```

VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_slli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW:__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW:__m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW:__m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW:__m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW:__m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW:__m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD:__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD:__m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD:__m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD:__m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD:__m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD:__m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ:__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ:__m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ:__m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ:__m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ:__m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ:__m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.

Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.

Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E1 /r[1] PSRAW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /r PSRAW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
NP 0F 71 /4 ib[1] PSRAW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib PSRAW *xmm1*, imm8	B	V/V	SSE2	Shift words in *xmm1* right by imm8 while shifting in sign bits
NP 0F E2 /r[1] PSRAD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /r PSRAD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
NP 0F 72 /4 ib[1] PSRAD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib PSRAD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits.
VEX.128.66.0F.WIG E1 /r VPSRAW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.128.66.0F.WIG 71 /4 ib VPSRAW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in sign bits.
VEX.128.66.0F.WIG E2 /r VPSRAD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.128.66.0F.WIG 72 /4 ib VPSRAD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in sign bits.
VEX.256.66.0F.WIG E1 /r VPSRAW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.256.66.0F.WIG 71 /4 ib VPSRAW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in sign bits.
VEX.256.66.0F.WIG E2 /r VPSRAD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.256.66.0F.WIG 72 /4 ib VPSRAD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in sign bits.
EVEX.128.66.0F.WIG E1 /r VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG E1 /r VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.

EVEX.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

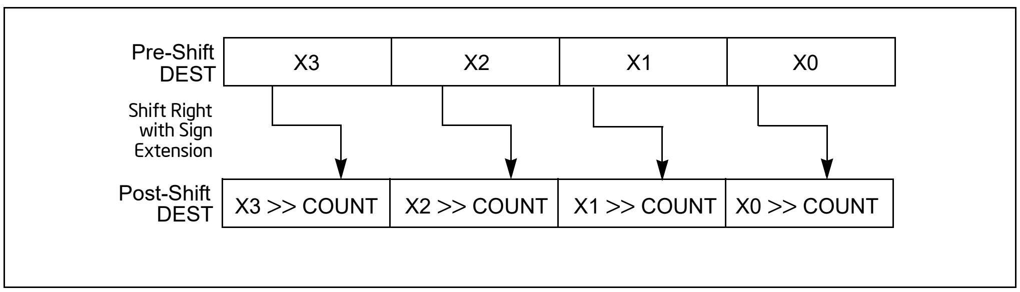

Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRAW (with 64-bit operand)

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

PSRAD (with 64-bit operand)

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

ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← SignBit
ELSE
    DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← SignBit
ELSE
    DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);
```

```
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL)                    ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT ← 64;
FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);


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

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAW (ymm, ymm, xmm/m128) - VEX
```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0
```

VPSRAW (ymm, imm8) - VEX
```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0
```

VPSRAW (xmm, xmm, xmm/m128) - VEX
```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

VPSRAW (xmm, imm8) - VEX
```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0
```

PSRAW (xmm, xmm, xmm/m128)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)
```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAD (EVEX versions, imm8)
```
(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) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAD (EVEX versions, xmm/m128)
```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
```

```
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0
```

VPSRAD (ymm, imm8) - VEX

```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0
```

VPSRAD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRAD (xmm, imm8) - VEX

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAQ (EVEX versions, imm8)

```
(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 (SRC1 *is memory*)
                THEN DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+63:i] ← 0
```

```
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAQ (EVEX versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] ← ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i _mm512_maskz_srai_epi64( __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i _mm512_maskz_sra_epi64( __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64( __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
```

VPSRAW __m512i _mm512_maskz_srai_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i _mm_maskz_srai_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i _mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i _mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i _mm512_maskz_sra_epi16(__mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i _mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i _mm256_maskz_sra_epi16(__mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i _mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i _mm_maskz_sra_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRAW:__m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW:__m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW:__m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW:__m128i _mm_sra_epi16(__m128i m, __m128i count)
VPSRAW:__m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW:__m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD:__m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD:__m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD:__m128i _mm_srai_epi32 (__m128i m, int count)
(V)PSRAD:__m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD:__m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD:__m256i _mm256_sra_epi32 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.

Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRAW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:

Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.

Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ *xmm1*, *imm8*	A	V/V	SSE2	Shift *xmm1* right by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG 73 /3 ib VPSRLDQ *xmm1, xmm2, imm8*	B	V/V	AVX	Shift *xmm2* right by *imm8* bytes while shifting in 0s.
VEX.256.66.0F.WIG 73 /3 ib VPSRLDQ *ymm1, ymm2, imm8*	B	V/V	AVX2	Shift *ymm1* right by *imm8* bytes while shifting in 0s.
EVEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8	C	V/V	AVX512VL AVX512BW	Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8	C	V/V	AVX512VL AVX512BW	Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.512.66.0F.WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW	Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r, w)	imm8	NA	NA
B	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA

Description

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

In 64-bit mode and not encoded with VEX/EVEX, 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 source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

VPSRLDQ (EVEX.512 encoded version)

TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
DEST[383:256] ← SRC[383:256] >> (TEMP * 8)
DEST[511:384] ← SRC[511:384] >> (TEMP * 8)
DEST[MAXVL-1:512] ← 0;

VPSRLDQ (VEX.256 and EVEX.256 encoded version)

TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
DEST[MAXVL-1:256] ← 0;

VPSRLDQ (VEX.128 and EVEX.128 encoded version)

TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[MAXVL-1:128] ← 0;

PSRLDQ(128-bit Legacy SSE version)

TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents

(V)PSRLDQ __m128i _mm_srli_si128 (__m128i a, int imm)
VPSRLDQ __m256i _mm256_bsrli_epi128 (__m256i, const int)
VPSRLDQ __m512i _mm512_bsrli_epi128 (__m512i, int)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /r PSRLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib PSRLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D2 /r[1] PSRLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /r PSRLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib PSRLD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D3 /r[1] PSRLQ *mm, mm/m64*	A	V/V	MMX	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /r PSRLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ *mm, imm8*	B	V/V	MMX	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib PSRLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG D1 /r VPSRLW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.128.66.0F.WIG 71 /2 ib VPSRLW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG D2 /r VPSRLD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.128.66.0F.WIG 72 /2 ib VPSRLD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in 0s.
VEX.128.66.0F.WIG D3 /r VPSRLQ *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift quadwords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.128.66.0F.WIG 73 /2 ib VPSRLQ *xmm1, xmm2, imm8*	D	V/V	AVX	Shift quadwords in *xmm2* right by *imm8* while shifting in 0s.
VEX.256.66.0F.WIG D1 /r VPSRLW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.256.66.0F.WIG 71 /2 ib VPSRLW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in 0s.

VEX.256.66.0F.WIG D2 /r VPSRLD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.256.66.0F.WIG 72 /2 ib VPSRLD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in 0s.
VEX.256.66.0F.WIG D3 /r VPSRLQ *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift quadwords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.256.66.0F.WIG 73 /2 ib VPSRLQ *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift quadwords in *ymm2* right by *imm8* while shifting in 0s.
EVEX.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

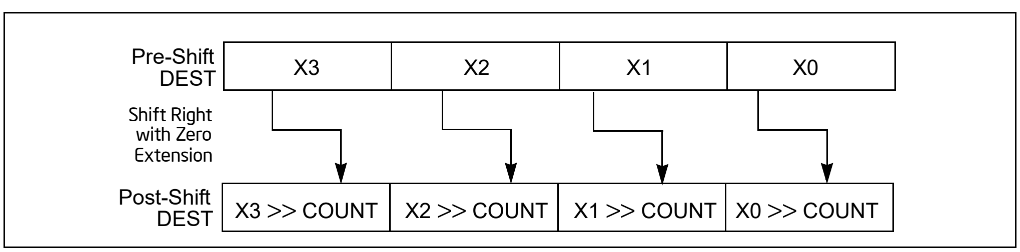

Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (with 64-bit operand)

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

PSRLD (with 64-bit operand)

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

PSRLQ (with 64-bit operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST ← ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
```

```
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ←0
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ←ZeroExtend(SRC[255:240] >> COUNT);
FI;

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

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ←0
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] >> COUNT);
FI;
```

```
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ←0
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ←ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLD (EVEX versions, imm8)

```
(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) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLD (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLD (xmm, imm8) - VEX.128 encoding
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLD (xmm, xmm, xmm/m128)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)
```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLQ (EVEX versions, xmm/m128)
```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLQ (EVEX versions, imm8)

```
(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 (SRC1 *is memory*)
                THEN DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32( __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m256i _mm256_maskz_srli_epi32( __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32( __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32( __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
```

VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW:__m64 _mm_srli_pi16(__m64 m, int count)
PSRLW:__m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW:__m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW:__m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW:__m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW:__m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD:__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD:__m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD:__m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD:__m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD:__m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD:__m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ:__m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ:__m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ:__m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ:__m128i _mm_srl_epi64 (__m128i m, __m128i count)
VPSRLQ:__m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ:__m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:
Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.
Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:
Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.
Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r[1] PSUBB *mm, mm/m64*	A	V/V	MMX	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm.*
66 0F F8 /r PSUBB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
NP 0F F9 /r[1] PSUBW *mm, mm/m64*	A	V/V	MMX	Subtract packed word integers in *mm/m64* from packed word integers in *mm.*
66 0F F9 /r PSUBW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
NP 0F FA /r[1] PSUBD *mm, mm/m64*	A	V/V	MMX	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm.*
66 0F FA /r PSUBD *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.
VEX.128.66.0F.WIG F8 /r VPSUBB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed byte integers in *xmm3/m128* from *xmm2.*
VEX.128.66.0F.WIG F9 /r VPSUBW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed word integers in *xmm3/m128* from *xmm2.*
VEX.128.66.0F.WIG FA /r VPSUBD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed doubleword integers in *xmm3/m128* from *xmm2.*
VEX.256.66.0F.WIG F8 /r VPSUBB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed byte integers in *ymm3/m256* from *ymm2.*
VEX.256.66.0F.WIG F9 /r VPSUBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed word integers in *ymm3/m256* from *ymm2.*
VEX.256.66.0F.WIG FA /r VPSUBD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed doubleword integers in *ymm3/m256* from *ymm2.*
EVEX.128.66.0F.WIG F8 /r VPSUBB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F8 /r VPSUBB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F8 /r VPSUBB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG F9 /r VPSUBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG F9 /r VPSUBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG F9 /r VPSUBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.

EVEX.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

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

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

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

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] − SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] − SRC[63:56];
```

PSUBW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] − SRC[15:0];
(* Repeat subtract operation for 2nd and 3rd word *)
DEST[63:48] ← DEST[63:48] − SRC[63:48];
```

PSUBD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] − SRC[31:0];
DEST[63:32] ← DEST[63:32] − SRC[63:32];
```

PSUBD (with 128-bit operands)

```
DEST[31:0]  ← DEST[31:0] − SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] ← DEST[127:96] − SRC[127:96];
```

VPSUBB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] - SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] - SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
```

```
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBD (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* 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*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBB (VEX.256 encoded version)

```
DEST[7:0] ←SRC1[7:0]-SRC2[7:0]
DEST[15:8] ←SRC1[15:8]-SRC2[15:8]
DEST[23:16] ←SRC1[23:16]-SRC2[23:16]
DEST[31:24] ←SRC1[31:24]-SRC2[31:24]
DEST[39:32] ←SRC1[39:32]-SRC2[39:32]
DEST[47:40] ←SRC1[47:40]-SRC2[47:40]
DEST[55:48] ←SRC1[55:48]-SRC2[55:48]
DEST[63:56] ←SRC1[63:56]-SRC2[63:56]
DEST[71:64] ←SRC1[71:64]-SRC2[71:64]
DEST[79:72] ←SRC1[79:72]-SRC2[79:72]
DEST[87:80] ←SRC1[87:80]-SRC2[87:80]
DEST[95:88] ←SRC1[95:88]-SRC2[95:88]
DEST[103:96] ←SRC1[103:96]-SRC2[103:96]
DEST[111:104] ←SRC1[111:104]-SRC2[111:104]
DEST[119:112] ←SRC1[119:112]-SRC2[119:112]
DEST[127:120] ←SRC1[127:120]-SRC2[127:120]
DEST[135:128] ←SRC1[135:128]-SRC2[135:128]
DEST[143:136] ←SRC1[143:136]-SRC2[143:136]
DEST[151:144] ←SRC1[151:144]-SRC2[151:144]
DEST[159:152] ←SRC1[159:152]-SRC2[159:152]
DEST[167:160] ←SRC1[167:160]-SRC2[167:160]
DEST[175:168] ←SRC1[175:168]-SRC2[175:168]
DEST[183:176] ←SRC1[183:176]-SRC2[183:176]
DEST[191:184] ←SRC1[191:184]-SRC2[191:184]
DEST[199:192] ←SRC1[199:192]-SRC2[199:192]
DEST[207:200] ←SRC1[207:200]-SRC2[207:200]
```

DEST[215:208] ←SRC1[215:208]-SRC2[215:208]
DEST[223:216] ←SRC1[223:216]-SRC2[223:216]
DEST[231:224] ←SRC1[231:224]-SRC2[231:224]
DEST[239:232] ←SRC1[239:232]-SRC2[239:232]
DEST[247:240] ←SRC1[247:240]-SRC2[247:240]
DEST[255:248] ←SRC1[255:248]-SRC2[255:248]
DEST[MAXVL-1:256] ←0

VPSUBB (VEX.128 encoded version)
DEST[7:0] ←SRC1[7:0]-SRC2[7:0]
DEST[15:8] ←SRC1[15:8]-SRC2[15:8]
DEST[23:16] ←SRC1[23:16]-SRC2[23:16]
DEST[31:24] ←SRC1[31:24]-SRC2[31:24]
DEST[39:32] ←SRC1[39:32]-SRC2[39:32]
DEST[47:40] ←SRC1[47:40]-SRC2[47:40]
DEST[55:48] ←SRC1[55:48]-SRC2[55:48]
DEST[63:56] ←SRC1[63:56]-SRC2[63:56]
DEST[71:64] ←SRC1[71:64]-SRC2[71:64]
DEST[79:72] ←SRC1[79:72]-SRC2[79:72]
DEST[87:80] ←SRC1[87:80]-SRC2[87:80]
DEST[95:88] ←SRC1[95:88]-SRC2[95:88]
DEST[103:96] ←SRC1[103:96]-SRC2[103:96]
DEST[111:104] ←SRC1[111:104]-SRC2[111:104]
DEST[119:112] ←SRC1[119:112]-SRC2[119:112]
DEST[127:120] ←SRC1[127:120]-SRC2[127:120]
DEST[MAXVL-1:128] ←0

PSUBB (128-bit Legacy SSE version)
DEST[7:0] ←DEST[7:0]-SRC[7:0]
DEST[15:8] ←DEST[15:8]-SRC[15:8]
DEST[23:16] ←DEST[23:16]-SRC[23:16]
DEST[31:24] ←DEST[31:24]-SRC[31:24]
DEST[39:32] ←DEST[39:32]-SRC[39:32]
DEST[47:40] ←DEST[47:40]-SRC[47:40]
DEST[55:48] ←DEST[55:48]-SRC[55:48]
DEST[63:56] ←DEST[63:56]-SRC[63:56]
DEST[71:64] ←DEST[71:64]-SRC[71:64]
DEST[79:72] ←DEST[79:72]-SRC[79:72]
DEST[87:80] ←DEST[87:80]-SRC[87:80]
DEST[95:88] ←DEST[95:88]-SRC[95:88]
DEST[103:96] ←DEST[103:96]-SRC[103:96]
DEST[111:104] ←DEST[111:104]-SRC[111:104]
DEST[119:112] ←DEST[119:112]-SRC[119:112]
DEST[127:120] ←DEST[127:120]-SRC[127:120]
DEST[MAXVL-1:128] (Unmodified)

VPSUBW (VEX.256 encoded version)
DEST[15:0] ←SRC1[15:0]-SRC2[15:0]
DEST[31:16] ←SRC1[31:16]-SRC2[31:16]
DEST[47:32] ←SRC1[47:32]-SRC2[47:32]
DEST[63:48] ←SRC1[63:48]-SRC2[63:48]
DEST[79:64] ←SRC1[79:64]-SRC2[79:64]
DEST[95:80] ←SRC1[95:80]-SRC2[95:80]
DEST[111:96] ←SRC1[111:96]-SRC2[111:96]

DEST[127:112] ←SRC1[127:112]-SRC2[127:112]
DEST[143:128] ←SRC1[143:128]-SRC2[143:128]
DEST[159:144] ←SRC1[159:144]-SRC2[159:144]
DEST[175:160] ←SRC1[175:160]-SRC2[175:160]
DEST[191:176] ←SRC1[191:176]-SRC2[191:176]
DEST[207:192] ←SRC1207:192]-SRC2[207:192]
DEST[223:208] ←SRC1[223:208]-SRC2[223:208]
DEST[239:224] ←SRC1[239:224]-SRC2[239:224]
DEST[255:240] ←SRC1[255:240]-SRC2[255:240]
DEST[MAXVL-1:256] ←0

VPSUBW (VEX.128 encoded version)
DEST[15:0] ←SRC1[15:0]-SRC2[15:0]
DEST[31:16] ←SRC1[31:16]-SRC2[31:16]
DEST[47:32] ←SRC1[47:32]-SRC2[47:32]
DEST[63:48] ←SRC1[63:48]-SRC2[63:48]
DEST[79:64] ←SRC1[79:64]-SRC2[79:64]
DEST[95:80] ←SRC1[95:80]-SRC2[95:80]
DEST[111:96] ←SRC1[111:96]-SRC2[111:96]
DEST[127:112] ←SRC1[127:112]-SRC2[127:112]
DEST[MAXVL-1:128] ←0

PSUBW (128-bit Legacy SSE version)
DEST[15:0] ←DEST[15:0]-SRC[15:0]
DEST[31:16] ←DEST[31:16]-SRC[31:16]
DEST[47:32] ←DEST[47:32]-SRC[47:32]
DEST[63:48] ←DEST[63:48]-SRC[63:48]
DEST[79:64] ←DEST[79:64]-SRC[79:64]
DEST[95:80] ←DEST[95:80]-SRC[95:80]
DEST[111:96] ←DEST[111:96]-SRC[111:96]
DEST[127:112] ←DEST[127:112]-SRC[127:112]
DEST[MAXVL-1:128] (Unmodified)

VPSUBD (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

VPSUBD (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

PSUBD (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)

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBW __m512i _mm512_sub_epi16(__m512i a, __m512i b);
VPSUBW __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBW __m512i _mm512_maskz_sub_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBW __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBW __m256i _mm256_maskz_sub_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBW __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBW __m128i _mm_maskz_sub_epi16(__mmask8 k, __m128i a, __m128i b);
VPSUBD __m512i _mm512_sub_epi32(__m512i a, __m512i b);
VPSUBD __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBD __m512i _mm512_maskz_sub_epi32(__mmask16 k, __m512i a, __m512i b);
VPSUBD __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBD __m256i _mm256_maskz_sub_epi32(__mmask8 k, __m256i a, __m256i b);
VPSUBD __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBD __m128i _mm_maskz_sub_epi32(__mmask8 k, __m128i a, __m128i b);
PSUBB:__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
(V)PSUBB:__m128i _mm_sub_epi8 (__m128i a, __m128i b)
VPSUBB:__m256i _mm256_sub_epi8 (__m256i a, __m256i b)
PSUBW:__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
(V)PSUBW:__m128i _mm_sub_epi16 (__m128i a, __m128i b)
VPSUBW:__m256i _mm256_sub_epi16 (__m256i a, __m256i b)
PSUBD:__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD:__m128i _mm_sub_epi32 (__m128i a, __m128i b)
VPSUBD:__m256i _mm256_sub_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBD, see Exceptions Type E4.

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb.

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r[1] PSUBQ *mm1, mm2/m64*	A	V/V	SSE2	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /r PSUBQ *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.
VEX.128.66.0F.WIG FB/r VPSUBQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed quadword integers in *xmm3/m128* from *xmm2*.
VEX.256.66.0F.WIG FB /r VPSUBQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed quadword integers in *ymm3/m256* from *ymm2.*
EVEX.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBQ (with 64-Bit operands)

```
    DEST[63:0] ← DEST[63:0] − SRC[63:0];
```

PSUBQ (with 128-Bit operands)

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

VPSUBQ (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
```

VPSUBQ (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
```

VPSUBQ (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] - 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*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ:__m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ:__m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ:__m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBQ, see Exceptions Type E4.

PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /*r*[1] PSUBSB *mm, mm/m64*	A	V/V	MMX	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results.
66 0F E8 /*r* PSUBSB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
NP 0F E9 /*r*[1] PSUBSW *mm, mm/m64*	A	V/V	MMX	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results.
66 0F E9 /*r* PSUBSW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results.
VEX.128.66.0F.WIG E8 /r VPSUBSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed signed byte integers in *xmm3/m128* from packed signed byte integers in *xmm2* and saturate results.
VEX.128.66.0F.WIG E9 /r VPSUBSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed signed word integers in *xmm3/m128* from packed signed word integers in *xmm2* and saturate results.
VEX.256.66.0F.WIG E8 /r VPSUBSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed signed byte integers in *ymm3/m256* from packed signed byte integers in *ymm2* and saturate results.
VEX.256.66.0F.WIG E9 /r VPSUBSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed signed word integers in *ymm3/m256* from packed signed word integers in *ymm2* and saturate results.
EVEX.128.66.0F.WIG E8 /r VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E8 /r VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG E8 /r VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG E9 /r VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG E9 /r VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.

EVEX.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

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

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

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSW (with 64-bit operands)

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

VPSUBSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSUBSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] ←0;
```

VPSUBSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] ← 0;
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified);
```

VPSUBSW (VEX.256 encoded version)

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] ← 0;

VPSUBSW (VEX.128 encoded version)

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] ← 0;

PSUBSW (128-bit Legacy SSE Version)

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified);

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm_maskz_subs_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm_maskz_subs_epi16(__mmask8 k, __m128i a, __m128i b);
PSUBSB:__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB:__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB:__m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW:__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW:__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW:__m256i _mm256_subs_epi16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D8 /*r*[1] PSUBUSB *mm, mm/m64*	A	V/V	MMX	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /*r* PSUBUSB *xmm1*, *xmm2/m128*	A	V/V	SSE2	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
NP 0F D9 /*r*[1] PSUBUSW *mm, mm/m64*	A	V/V	MMX	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /*r* PSUBUSW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in *xmm1* and saturate result.
VEX.128.66.0F.WIG D8 /r VPSUBUSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed unsigned byte integers in *xmm3/m128* from packed unsigned byte integers in *xmm2* and saturate result.
VEX.128.66.0F.WIG D9 /r VPSUBUSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed unsigned word integers in *xmm3/m128* from packed unsigned word integers in *xmm2* and saturate result.
VEX.256.66.0F.WIG D8 /r VPSUBUSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed unsigned byte integers in *ymm3/m256* from packed unsigned byte integers in *ymm2* and saturate result.
VEX.256.66.0F.WIG D9 /r VPSUBUSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed unsigned word integers in *ymm3/m256* from packed unsigned word integers in *ymm2* and saturate result.
EVEX.128.66.0F.WIG D8 /r VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D8 /r VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.512.66.0F.WIG D8 /r VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.
EVEX.128.66.0F.WIG D9 /r VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.256.66.0F.WIG D9 /r VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.

EVEX.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] − SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] − SRC[63:56];
```

PSUBUSW (with 64-bit operands)
```
    DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] − SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] − SRC[63:48] );
```

VPSUBUSB (EVEX encoded versions)
```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSUBUSW (EVEX encoded versions)
```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16;
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSUBUSB (VEX.256 encoded version)
```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] ← 0;
```

VPSUBUSB (VEX.128 encoded version)
```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

PSUBUSB (128-bit Legacy SSE Version)
```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBUSW (VEX.256 encoded version)

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] ← 0;

VPSUBUSW (VEX.128 encoded version)

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] ← 0

PSUBUSW (128-bit Legacy SSE Version)

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSW __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSW __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m512i _mm512_maskz_subs_epu16(__mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m256i _mm256_mask_subs_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m256i _mm256_maskz_subs_epu16(__mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSW __m128i _mm_maskz_subs_epu16(__mmask8 k, __m128i a, __m128i b);
PSUBUSB:__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
(V)PSUBUSB:__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
VPSUBUSB:__m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSW:__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
(V)PSUBUSW:__m128i _mm_subs_epu16(__m128i m1, __m128i m2)
VPSUBUSW:__m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PTEST- Logical Compare

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Set ZF if *xmm2/m128* AND *xmm1* result is all 0s. Set CF if *xmm2/m128* AND NOT *xmm1* result is all 0s.
VEX.128.66.0F38.WIG 17 /r VPTEST *xmm1, xmm2/m128*	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.
VEX.256.66.0F38.WIG 17 /r VPTEST *ymm1, ymm2/m256*	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.

Instruction Operand Encoding

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

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

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

Operation

(V)PTEST (128-bit version)

```
IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VPTEST (VEX.256 encoded version)

```
IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

PTEST

int _mm_testz_si128 (__m128i s1, __m128i s2);

int _mm_testc_si128 (__m128i s1, __m128i s2);

int _mm_testnzc_si128 (__m128i s1, __m128i s2);

VPTEST

int _mm256_testz_si256 (__m256i s1, __m256i s2);

int _mm256_testc_si256 (__m256i s1, __m256i s2);

int _mm256_testnzc_si256 (__m256i s1, __m256i s2);

int _mm_testz_si128 (__m128i s1, __m128i s2);

int _mm_testc_si128 (__m128i s1, __m128i s2);

int _mm_testnzc_si128 (__m128i s1, __m128i s2);

Flags Affected

The 0F, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

PTWRITE - Write Data to a Processor Trace Packet

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
F3 REX.W 0F AE /4 PTWRITE *r64/m64*	RM	V/N.E		Reads the data from r64/m64 to encode into a PTW packet if dependencies are met (see details below).
F3 0F AE /4 PTWRITE *r32/m32*	RM	V/V		Reads the data from r32/m32 to encode into a PTW packet if dependencies are met (see details below).

Instruction Operand Encoding

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

Description

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, Section 35.2.2, "Software Trace Instrumentation with PTWRITE". The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

Note: The instruction will #UD if prefix 66H is used.

Operation

```
IF (IA32_RTIT_STATUS.TriggerEn & IA32_RTIT_STATUS.ContextEn & IA32_RTIT_STATUS.FilterEn & IA32_RTIT_CTL.PTWEn) = 1
    PTW.PayloadBytes ← Encoded payload size;
    PTW.IP ← IA32_RTIT_CTL.FUPonPTW
    IF IA32_RTIT_CTL.FUPonPTW = 1
        Insert FUP packet with IP of PTWRITE;
    FI;
FI;
```

Flags Affected

None.

Other 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 an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H 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.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H 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 an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H 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.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 68 /r[1] PUNPCKHBW *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm.*
66 0F 68 /r PUNPCKHBW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1.*
NP 0F 69 /r[1] PUNPCKHWD *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order words from *mm* and *mm/m64* into *mm.*
66 0F 69 /r PUNPCKHWD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1.*
NP 0F 6A /r[1] PUNPCKHDQ *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm.*
66 0F 6A /r PUNPCKHDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1.*
66 0F 6D /r PUNPCKHQDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1.*
VEX.128.66.0F.WIG 68/r VPUNPCKHBW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order bytes from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.128.66.0F.WIG 69/r VPUNPCKHWD *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order words from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.128.66.0F.WIG 6A/r VPUNPCKHDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order doublewords from *xmm2* and *xmm3/m128* into *xmm1.*
VEX.128.66.0F.WIG 6D/r VPUNPCKHQDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.256.66.0F.WIG 68 /r VPUNPCKHBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.256.66.0F.WIG 69 /r VPUNPCKHWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.256.66.0F.WIG 6A /r VPUNPCKHDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.256.66.0F.WIG 6D /r VPUNPCKHQDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.
EVEX.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.128.66.0F.W0 6A /r VPUNPCKHDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.
EVEX.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.

EVEX.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

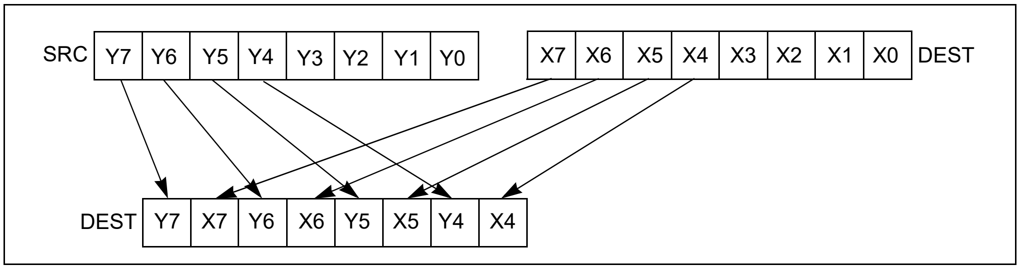

Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands

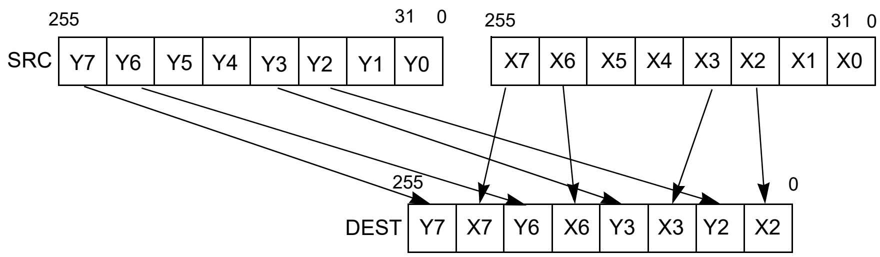

Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW instruction with 64-bit operands:

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

PUNPCKHW instruction with 64-bit operands:

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

PUNPCKHDQ instruction with 64-bit operands:

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

```
INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
```

DEST[167:160] ← SRC1[215:208]
DEST[175:168] ← SRC2[215:208]
DEST[183:176] ← SRC1[223:216]
DEST[191:184] ← SRC2[223:216]
DEST[199:192] ← SRC1[231:224]
DEST[207:200] ← SRC2[231:224]
DEST[215:208] ← SRC1[239:232]
DEST[223:216] ← SRC2[239:232]
DEST[231:224] ← SRC1[247:240]
DEST[239:232] ← SRC2[247:240]
DEST[247:240] ← SRC1[255:248]
DEST[255:248] ← SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]
DEST[143:128] ← SRC1[207:192]
DEST[159:144] ← SRC2[207:192]
DEST[175:160] ← SRC1[223:208]
DEST[191:176] ← SRC2[223:208]
DEST[207:192] ← SRC1[239:224]
DEST[223:208] ← SRC2[239:224]
DEST[239:224] ← SRC1[255:240]
DEST[255:240] ← SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128] ← SRC1[255:192]
DEST[255:192] ← SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW (128-bit Legacy SSE Version)
DEST[127:0] ←INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHBW (VEX.128 encoded version)
DEST[127:0] ←INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] ←0

VPUNPCKHBW (VEX.256 encoded version)
DEST[255:0] ←INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0

VPUNPCKHBW (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

PUNPCKHWD (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHWD (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ←0
```

VPUNPCKHWD (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKHWD (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

PUNPCKHDQ (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHDQ (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ←0
```

VPUNPCKHDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKHDQ (EVEX.512 encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] ← 0
```

PUNPCKHQDQ (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKHQDQ (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPUNPCKHQDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKHQDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_mask_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8( __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m128i _mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);
```

VPUNPCKHBW __m128i _mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKHWD __m512i _mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i _mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m128i _mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHWD __m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i _mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_mask_unpackhi_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
PUNPCKHBW:__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW:__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW:__m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
PUNPCKHWD:__m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
(V)PUNPCKHWD:__m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD:__m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)
PUNPCKHDQ:__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ:__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
VPUNPCKHDQ:__m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
(V)PUNPCKHQDQ:__m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
VPUNPCKHQDQ:__m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 60 /r[1] PUNPCKLBW *mm, mm/m32*	A	V/V	MMX	Interleave low-order bytes from *mm* and *mm/m32* into *mm*.
66 0F 60 /r PUNPCKLBW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
NP 0F 61 /r[1] PUNPCKLWD *mm, mm/m32*	A	V/V	MMX	Interleave low-order words from *mm* and *mm/m32* into *mm*.
66 0F 61 /r PUNPCKLWD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*.
NP 0F 62 /r[1] PUNPCKLDQ *mm, mm/m32*	A	V/V	MMX	Interleave low-order doublewords from *mm* and *mm/m32* into *mm*.
66 0F 62 /r PUNPCKLDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1.
66 0F 6C /r PUNPCKLQDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* register.
VEX.128.66.0F.WIG 60/r VPUNPCKLBW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order bytes from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.128.66.0F.WIG 61/r VPUNPCKLWD *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order words from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.128.66.0F.WIG 62/r VPUNPCKLDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order doublewords from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.128.66.0F.WIG 6C/r VPUNPCKLQDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.256.66.0F.WIG 60 /r VPUNPCKLBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.256.66.0F.WIG 61 /r VPUNPCKLWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.256.66.0F.WIG 62 /r VPUNPCKLDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.256.66.0F.WIG 6C /r VPUNPCKLQDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.
EVEX.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W0 62 /r VPUNPCKLDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.
EVEX.128.66.0F.W1 6C /r VPUNPCKLQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

EVEX.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

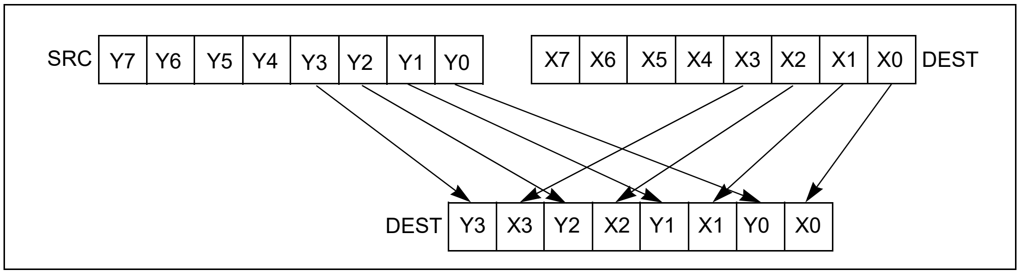

Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

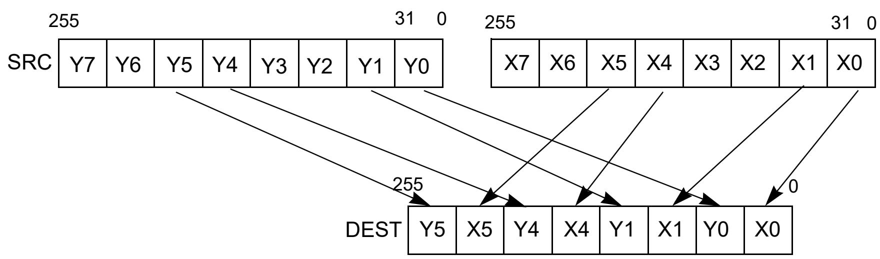

Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

```
PUNPCKLBW instruction with 64-bit operands:
    DEST[63:56] ← SRC[31:24];
    DEST[55:48] ← DEST[31:24];
    DEST[47:40] ← SRC[23:16];
    DEST[39:32] ← DEST[23:16];
    DEST[31:24] ← SRC[15:8];
    DEST[23:16] ← DEST[15:8];
    DEST[15:8] ← SRC[7:0];
    DEST[7:0] ← DEST[7:0];

PUNPCKLWD instruction with 64-bit operands:
    DEST[63:48] ← SRC[31:16];
    DEST[47:32] ← DEST[31:16];
    DEST[31:16] ← SRC[15:0];
    DEST[15:0] ← DEST[15:0];

PUNPCKLDQ instruction with 64-bit operands:
    DEST[63:32] ← SRC[31:0];
    DEST[31:0] ← DEST[31:0];
INTERLEAVE_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC1[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
DEST[135:128] ← SRC1[135:128]
DEST[143:136] ← SRC2[135:128]
DEST[151:144] ← SRC1[143:136]
DEST[159:152] ← SRC2[143:136]
DEST[167:160] ← SRC1[151:144]
```

```
DEST[175:168] ← SRC2[151:144]
DEST[183:176] ← SRC1[159:152]
DEST[191:184] ← SRC2[159:152]
DEST[199:192] ← SRC1[167:160]
DEST[207:200] ← SRC2[167:160]
DEST[215:208] ← SRC1[175:168]
DEST[223:216] ← SRC2[175:168]
DEST[231:224] ← SRC1[183:176]
DEST[239:232] ← SRC2[183:176]
DEST[247:240] ← SRC1[191:184]
DEST[255:248] ← SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
DEST[143:128] ← SRC1[143:128]
DEST[159:144] ← SRC2[143:128]
DEST[175:160] ← SRC1[159:144]
DEST[191:176] ← SRC2[159:144]
DEST[207:192] ← SRC1[175:160]
DEST[223:208] ← SRC2[175:160]
DEST[239:224] ← SRC1[191:176]
DEST[255:240] ← SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
```

DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

PUNPCKLBW
DEST[127:0] ←INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)
DEST[127:0] ←INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] ←0

VPUNPCKLBW (VEX.256 encoded instruction)
DEST[255:0] ←INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0

VPUNPCKLBW (EVEX.512 encoded instruction)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*              ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_BYTES_512b(SRC1, SRC2)
```

PUNPCKLWD

```
DEST[127:0] ←INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLWD (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ←0
```

VPUNPCKLWD (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKLWD (EVEX.512 encoded instruction)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_WORDS_512b(SRC1, SRC2)
```

PUNPCKLDQ

```
DEST[127:0] ←INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPUNPCKLDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKLDQ (EVEX encoded instructions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
```

```
ENDFOR
DEST511:0] ←INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] ← 0
```

PUNPCKLQDQ

```
DEST[127:0] ←INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPUNPCKLQDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKLQDQ (EVEX encoded instructions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
```

VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b);
PUNPCKLBW:__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW:__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW:__m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLWD:__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLWD:__m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD:__m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ:__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ:__m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ:__m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLQDQ:__m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLQDQ:__m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKLDQ/QDQ, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	M	Valid	Valid	Push *r/m16.*
FF /6	PUSH *r/m32*	M	N.E.	Valid	Push *r/m32.*
FF /6	PUSH *r/m64*	M	Valid	N.E.	Push *r/m64.*
50+*rw*	PUSH *r16*	O	Valid	Valid	Push *r16.*
50+*rd*	PUSH *r32*	O	N.E.	Valid	Push *r32.*
50+*rd*	PUSH *r64*	O	Valid	N.E.	Push *r64.*
6A ib	PUSH *imm8*	I	Valid	Valid	Push *imm8.*
68 iw	PUSH *imm16*	I	Valid	Valid	Push *imm16.*
68 id	PUSH *imm32*	I	Valid	Valid	Push *imm32.*
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

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)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when referencing a source operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.
- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

```
(* See Description section for possible sign-extension or zero-extension of source operand and for *)
(* a case in which the size of the memory store may be smaller than the instruction’s operand size *)
IF StackAddrSize = 64
    THEN
        IF OperandSize = 64
            THEN
                RSP ← RSP – 8;
                Memory[SS:RSP] ← SRC;           (* push quadword *)
        ELSE IF OperandSize = 32
            THEN
                RSP ← RSP – 4;
                Memory[SS:RSP] ← SRC;           (* push dword *)
            ELSE (* OperandSize = 16 *)
                RSP ← RSP – 2;
                Memory[SS:RSP] ← SRC;           (* push word *)
        FI;
ELSE IF StackAddrSize = 32
    THEN
        IF OperandSize = 64
            THEN
                ESP ← ESP – 8;
                Memory[SS:ESP] ← SRC;           (* push quadword *)
        ELSE IF OperandSize = 32
            THEN
                ESP ← ESP – 4;
                Memory[SS:ESP] ← SRC;           (* push dword *)
            ELSE (* OperandSize = 16 *)
                ESP ← ESP – 2;
                Memory[SS:ESP] ← SRC;           (* push word *)
        FI;
    ELSE (* StackAddrSize = 16 *)
```

```
        IF OperandSize = 32
            THEN
                SP ← SP - 4;
                Memory[SS:SP] ← SRC;            (* push dword *)
            ELSE (* OperandSize = 16 *)
                SP ← SP - 2;
                Memory[SS:SP] ← SRC;            (* push word *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

#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

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.
	If the PUSH is of CS, SS, DS, or ES.

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
60	PUSHA	ZO	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	ZO	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Instruction Operand Encoding

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

Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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

Operation

```
IF 64-bit Mode

    THEN #UD

FI;

IF OperandSize = 32 (* PUSHAD instruction *)
    THEN
        Temp ← (ESP);
        Push(EAX);
        Push(ECX);
        Push(EDX);
        Push(EBX);
        Push(Temp);
        Push(EBP);
        Push(ESI);
        Push(EDI);
    ELSE (* OperandSize = 16, PUSHA instruction *)
        Temp ← (SP);
        Push(AX);
        Push(CX);
        Push(DX);
```

```
        Push(BX);
        Push(Temp);
        Push(BP);
        Push(SI);
        Push(DI);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	ZO	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	ZO	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	ZO	Valid	N.E.	Push RFLAGS.

Instruction Operand Encoding

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

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFD instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Operation

```
IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
    THEN
        IF OperandSize = 32
            THEN
                push (EFLAGS AND 00FCFFFFH);
                (* VM and RF bits are cleared in image stored on the stack *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
```

```
        FI;
    ELSE IF 64-bit MODE (* In 64-bit Mode *)
        IF OperandSize = 64
            THEN
                push (RFLAGS AND 00000000_00FCFFFFH);
                (* VM and RF bits are cleared in image stored on the stack; *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
        FI;
    ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
        IF (CR4.VME = 0) OR (OperandSize = 32)
            THEN #GP(0); (* Trap to virtual-8086 monitor *)
            ELSE
                tempFLAGS = EFLAGS[15:0];
                tempFLAGS[9] = tempFLAGS[19];     (* VIF replaces IF *)
                tempFlags[13:12] = 3;  (* IOPL is set to 3 in image stored on the stack *)
                push (tempFLAGS);
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EF /r[1] PXOR *mm, mm/m64*	A	V/V	MMX	Bitwise XOR of *mm/m64* and *mm.*
66 0F EF /r PXOR *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise XOR of *xmm2/m128* and *xmm1.*
VEX.128.66.0F.WIG EF /r VPXOR *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise XOR of *xmm3/m128* and *xmm2.*
VEX.256.66.0F.WIG EF /r VPXOR *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Bitwise XOR of *ymm3/m256* and *ymm2.*
EVEX.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

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

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

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

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 register destination are zeroed.

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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PXOR (64-bit operand)

```
DEST ← DEST XOR SRC
```

PXOR (128-bit Legacy SSE version)

```
DEST ← DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPXOR (VEX.128 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[MAXVL-1:128] ← 0
```

VPXOR (VEX.256 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[MAXVL-1:256] ← 0
```

VPXORD (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* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPXORQ (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 XOR SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32( __mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32( __mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32( __mmask16 m, __m128i a, __m128i b)
VPXORQ __m512i _mm512_xor_epi64( __m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b);
VPXORQ __m256i _mm256_xor_epi64( __m256i a, __m256i b);
VPXORQ __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b);
VPXORQ __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b);
VPXORQ __m128i _mm_xor_epi64( __m128i a, __m128i b);
VPXORQ __m128i _mm_mask_xor_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPXORQ __m128i _mm_maskz_xor_epi64(__mmask8 m, __m128i a, __m128i b);
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 ( __m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

RCL/RCR/ROL/ROR—Rotate

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /2	RCL *r/m8*, 1	M1	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left once.
REX + D0 /2	RCL *r/m8**, 1	M1	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left once.
D2 /2	RCL *r/m8*, CL	MC	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left CL times.
REX + D2 /2	RCL *r/m8**, CL	MC	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left CL times.
C0 /2 *ib*	RCL *r/m8, imm8*	MI	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
REX + C0 /2 *ib*	RCL *r/m8*, imm8*	MI	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
D1 /2	RCL *r/m16*, 1	M1	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left once.
D3 /2	RCL *r/m16*, CL	MC	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left CL times.
C1 /2 *ib*	RCL *r/m16, imm8*	MI	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left *imm8* times.
D1 /2	RCL *r/m32*, 1	M1	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left once.
REX.W + D1 /2	RCL *r/m64*, 1	M1	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left once. Uses a 6 bit count.
D3 /2	RCL *r/m32*, CL	MC	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left CL times.
REX.W + D3 /2	RCL *r/m64*, CL	MC	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left CL times. Uses a 6 bit count.
C1 /2 *ib*	RCL *r/m32, imm8*	MI	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left *imm8* times.
REX.W + C1 /2 *ib*	RCL *r/m64, imm8*	MI	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left *imm8* times. Uses a 6 bit count.
D0 /3	RCR *r/m8*, 1	M1	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right once.
REX + D0 /3	RCR *r/m8**, 1	M1	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right once.
D2 /3	RCR *r/m8*, CL	MC	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right CL times.
REX + D2 /3	RCR *r/m8**, CL	MC	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right CL times.
C0 /3 *ib*	RCR *r/m8, imm8*	MI	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
REX + C0 /3 *ib*	RCR *r/m8*, imm8*	MI	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
D1 /3	RCR *r/m16*, 1	M1	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right once.
D3 /3	RCR *r/m16*, CL	MC	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right CL times.
C1 /3 *ib*	RCR *r/m16, imm8*	MI	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right *imm8* times.
D1 /3	RCR *r/m32*, 1	M1	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR *r/m64*, 1	M1	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right once. Uses a 6 bit count.
D3 /3	RCR *r/m32*, CL	MC	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right CL times.
REX.W + D3 /3	RCR *r/m64*, CL	MC	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right CL times. Uses a 6 bit count.
C1 /3 *ib*	RCR *r/m32, imm8*	MI	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right *imm8* times.
REX.W + C1 /3 *ib*	RCR *r/m64, imm8*	MI	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right *imm8* times. Uses a 6 bit count.
D0 /0	ROL *r/m8*, 1	M1	Valid	Valid	Rotate 8 bits *r/m8* left once.
REX + D0 /0	ROL *r/m8**, 1	M1	Valid	N.E.	Rotate 8 bits *r/m8* left once
D2 /0	ROL *r/m8*, CL	MC	Valid	Valid	Rotate 8 bits *r/m8* left CL times.
REX + D2 /0	ROL *r/m8**, CL	MC	Valid	N.E.	Rotate 8 bits *r/m8* left CL times.
C0 /0 *ib*	ROL *r/m8, imm8*	MI	Valid	Valid	Rotate 8 bits *r/m8* left *imm8* times.

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + C0 /0 *ib*	ROL *r/m8*, imm8*	MI	Valid	N.E.	Rotate 8 bits *r/m8* left *imm8* times.
D1 /0	ROL *r/m16*, 1	M1	Valid	Valid	Rotate 16 bits *r/m16* left once.
D3 /0	ROL *r/m16*, CL	MC	Valid	Valid	Rotate 16 bits *r/m16* left CL times.
C1 /0 *ib*	ROL *r/m16, imm8*	MI	Valid	Valid	Rotate 16 bits *r/m16* left *imm8* times.
D1 /0	ROL *r/m32*, 1	M1	Valid	Valid	Rotate 32 bits *r/m32* left once.
REX.W + D1 /0	ROL *r/m64*, 1	M1	Valid	N.E.	Rotate 64 bits *r/m64* left once. Uses a 6 bit count.
D3 /0	ROL *r/m32*, CL	MC	Valid	Valid	Rotate 32 bits *r/m32* left CL times.
REX.W + D3 /0	ROL *r/m64*, CL	MC	Valid	N.E.	Rotate 64 bits *r/m64* left CL times. Uses a 6 bit count.
C1 /0 *ib*	ROL *r/m32, imm8*	MI	Valid	Valid	Rotate 32 bits *r/m32* left *imm8* times.
REX.W + C1 /0 *ib*	ROL *r/m64, imm8*	MI	Valid	N.E.	Rotate 64 bits *r/m64* left *imm8* times. Uses a 6 bit count.
D0 /1	ROR *r/m8*, 1	M1	Valid	Valid	Rotate 8 bits *r/m8* right once.
REX + D0 /1	ROR *r/m8**, 1	M1	Valid	N.E.	Rotate 8 bits *r/m8* right once.
D2 /1	ROR *r/m8*, CL	MC	Valid	Valid	Rotate 8 bits *r/m8* right CL times.
REX + D2 /1	ROR *r/m8**, CL	MC	Valid	N.E.	Rotate 8 bits *r/m8* right CL times.
C0 /1 *ib*	ROR *r/m8, imm8*	MI	Valid	Valid	Rotate 8 bits *r/m16* right *imm8* times.
REX + C0 /1 *ib*	ROR *r/m8*, imm8*	MI	Valid	N.E.	Rotate 8 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m16*, 1	M1	Valid	Valid	Rotate 16 bits *r/m16* right once.
D3 /1	ROR *r/m16*, CL	MC	Valid	Valid	Rotate 16 bits *r/m16* right CL times.
C1 /1 *ib*	ROR *r/m16, imm8*	MI	Valid	Valid	Rotate 16 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m32*, 1	M1	Valid	Valid	Rotate 32 bits *r/m32* right once.
REX.W + D1 /1	ROR *r/m64*, 1	M1	Valid	N.E.	Rotate 64 bits *r/m64* right once. Uses a 6 bit count.
D3 /1	ROR *r/m32*, CL	MC	Valid	Valid	Rotate 32 bits *r/m32* right CL times.
REX.W + D3 /1	ROR *r/m64*, CL	MC	Valid	N.E.	Rotate 64 bits *r/m64* right CL times. Uses a 6 bit count.
C1 /1 *ib*	ROR *r/m32, imm8*	MI	Valid	Valid	Rotate 32 bits *r/m32* right *imm8* times.
REX.W + C1 /1 *ib*	ROR *r/m64, imm8*	MI	Valid	N.E.	Rotate 64 bits *r/m64* right *imm8* times. Uses a 6 bit count.

NOTES:

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

** See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	NA	NA
MC	ModRM:r/m (w)	CL	NA	NA
MI	ModRM:r/m (w)	imm8	NA	NA

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
(* RCL and RCR instructions *)
SIZE ← OperandSize;
CASE (determine count) OF
    SIZE ← 8:     tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE ← 16:    tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE ← 32:    tempCOUNT ← COUNT AND 1FH;
    SIZE ← 64:    tempCOUNT ← COUNT AND 3FH;
ESAC;

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

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

(* ROL and ROR instructions *)
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE

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

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

Flags Affected

If the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined. The CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#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 source operand is located in a nonwritable segment.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
NP 0F 53 /r RCPPS *xmm1, xmm2/m128*	RM	V/V	SSE	Computes the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.0F.WIG 53 /r VRCPPS *xmm1, xmm2/m128*	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in *xmm2/mem* and stores the results in *xmm1*.
VEX.256.0F.WIG 53 /r VRCPPS *ymm1, ymm2/m256*	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in *ymm2/mem* and stores the results in *ymm1*.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

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.

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

Operation

RCPPS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

VRCPPS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] ← 0

VRCPPS (VEX.256 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
DEST[255:224] ← APPROXIMATE(1/SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

RCCPS: __m128 _mm_rcp_ps(__m128 a)

RCPPS: __m256 _mm256_rcp_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 53 /*r* RCPSS *xmm1*, xmm2/m32	RM	V/V	SSE	Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm2/m32* and stores the result in *xmm1*.
VEX.LIG.F3.0F.WIG 53 /r VRCPSS *xmm1, xmm2, xmm3/m32*	RVM	V/V	AVX	Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm3/m32* and stores the result in *xmm1*. Also, upper single precision floating-point values (bits[127:32]) from *xmm2* are copied to *xmm1*[127:32].

Instruction Operand Encoding

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

Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

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: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RCPSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

VRCPSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS: __m128 _mm_rcp_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

RDFSBASE/RDGSBASE—Read FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /0 RDFSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the FS base address.
F3 REX.W 0F AE /0 RDFSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the FS base address.
F3 0F AE /1 RDGSBASE *r32*	M	V/I	FSGSBASE	Load the 32-bit destination register with the GS base address.
F3 REX.W 0F AE /1 RDGSBASE *r64*	M	V/I	FSGSBASE	Load the 64-bit destination register with the GS base address.

Instruction Operand Encoding

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

Description

Loads the general-purpose register indicated by the modR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

Operation

DEST ← FS/GS segment base address;

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

RDFSBASE: unsigned int _readfsbase_u32(void);

RDFSBASE: unsigned __int64 _readfsbase_u64(void);

RDGSBASE: unsigned int _readgsbase_u32(void);

RDGSBASE: unsigned __int64 _readgsbase_u64(void);

Protected Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.FSGSBASE[bit 16] = 0.
	If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0.

RDMSR—Read from Model Specific Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 32	RDMSR	ZO	Valid	Valid	Read MSR specified by ECX into EDX:EAX.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

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

Operation

```
EDX:EAX ← MSR[ECX];
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The RDMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPID—Read Processor ID

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F C7 /7 RDPID r32	R	N.E./V	RDPID	Read IA32_TSC_AUX into r32.
F3 0F C7 /7 RDPID r64	R	V/N.E.	RDPID	Read IA32_TSC_AUX into r64.

Instruction Operand Encoding[1]

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

Description

Reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

Operation

DEST ← IA32_TSC_AUX

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.7H.0:ECX.RDPID[bit 22] = 0.

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.

1. ModRM.MOD = 011B required

RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EE	RDPKRU	ZO	V/V	OSPKE	Reads PKRU into EAX.

Instruction Operand Encoding

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

Description

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

Operation

```
IF (ECX = 0)
    THEN
        EAX ← PKRU;
        EDX ← 0;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDPKRU: uint32_t _rdpkru_u32(void);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

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.

RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 33	RDPMC	ZO	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

Instruction Operand Encoding

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

Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do no support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for “fast” reads.

The ECX register specifies the counter type (if the processor supports architectural performance monitoring) and counter index. Counter type is specified in ECX[30] to select one of two type of performance counters. If the processor does not support architectural performance monitoring, ECX[30:0] specifies the counter index; otherwise ECX[29:0] specifies the index relative to the base of each counter type. ECX[31] selects “fast” read mode if supported. The two counter types are:

- General-purpose or special-purpose performance counters are specified with ECX[30] = 0: The number of general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The availability of special-purpose counters, as well as the number of general-purpose counters if the processor does not support architectural performance monitoring, is model specific; see Chapter 18, “Performance Monitoring” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.
- Fixed-function performance counters are specified with ECX[30] = 1. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 18, “Performance Monitoring” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

The width of fixed-function performance counters and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, “Performance Monitoring Events,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

Operation

```
MSCB = Most Significant Counter Bit (* Model-specific *)

IF (((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0)) and (ECX indicates a supported counter))
    THEN
        EAX ← counter[31:0];
        EDX ← ZeroExtend(counter[MSCB:32]);
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified.
#UD	If the LOCK prefix is used.

RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NFx 0F C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
NFx 0F C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
NFx REX.W + 0F C7 /6 RDRAND r64	M	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

Instruction Operand Encoding

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

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 7.3.17, "Random Number Generator Instructions").

This instruction is available at all privilege levels.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B 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

```
IF HW_RND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_RND_GEN.data;
            osize is 32: DEST[31:0] ← HW_RND_GEN.data;
            osize is 16: DEST[15:0] ← HW_RND_GEN.data;
        ESAC
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC
        CF ← 0;
FI
OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

Intel C/C++ Compiler Intrinsic Equivalent

RDRAND: int _rdrand16_step(unsigned short *);

RDRAND: int _rdrand32_step(unsigned int *);

RDRAND: int _rdrand64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.01H:ECX.RDRAND[bit 30] = 0.

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.

RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NFx 0F C7 /7 RDSEED r16	M	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
NFx 0F C7 /7 RDSEED r32	M	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
NFx REX.W + 0F C7 /7 RDSEED r64	M	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

Instruction Operand Encoding

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

Description

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B 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

```
IF HW_NRND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_NRND_GEN.data;
            osize is 32: DEST[31:0] ← HW_NRND_GEN.data;
            osize is 16: DEST[15:0] ← HW_NRND_GEN.data;
        ESAC;
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC;
        CF ← 0;
FI;

OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

C/C++ Compiler Intrinsic Equivalent

RDSEED int _rdseed16_step(unsigned short *);

RDSEED int _rdseed32_step(unsigned int *);

RDSEED int _rdseed64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	ZO	Valid	Valid	Read time-stamp counter into EDX:EAX.

Instruction Operand Encoding

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

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSC:

- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads are globally visible,[1] it can execute LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads and stores are globally visible, it can execute the sequence MFENCE;LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute the sequence LFENCE immediately after RDTSC.

This instruction was introduced by the Pentium processor.

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

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

1. A load is considered to become globally visible when the value to be loaded is determined.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 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)	If the TSD flag in register CR4 is set.
#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.

RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F9	RDTSCP	ZO	Valid	Valid	Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.

Instruction Operand Encoding

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

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible.[1] But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSCP:

- If software requires RDTSCP to be executed only after all previous stores are globally visible, it can execute MFENCE immediately before RDTSCP.
- If software requires RDTSCP to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute LFENCE immediately after RDTSCP.

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

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX ← TimeStampCounter;
        ECX ← IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

1. A load is considered to become globally visible when the value to be loaded is determined.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS *m8*, DX	ZO	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS *m8*, DX	ZO	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS *m16*, DX	ZO	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI.]
F3 6D	REP INS *m32*, DX	ZO	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS *r/m32*, DX	ZO	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS *m8, m8*	ZO	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS *m8, m8*	ZO	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS *m16, m16*	ZO	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m32, m32*	ZO	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS *m64, m64*	ZO	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, *r/m8*	ZO	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, *r/m8**	ZO	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, *r/m16*	ZO	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, *r/m32*	ZO	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, *r/m32*	ZO	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	ZO	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS AL	ZO	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	ZO	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	ZO	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS RAX	ZO	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS *m8*	ZO	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS *m8*	ZO	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS *m16*	ZO	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS *m32*	ZO	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS *m64*	ZO	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS *m8, m8*	ZO	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS *m8, m8*	ZO	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS *m16, m16*	ZO	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m32, m32*	ZO	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS *m64, m64*	ZO	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS *m8*	ZO	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS *m8*	ZO	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS *m16*	ZO	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS *m32*	ZO	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 REX.W AF	REPE SCAS *m64*	ZO	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS *m8, m8*	ZO	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS *m8, m8*	ZO	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS *m16, m16*	ZO	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m32, m32*	ZO	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS *m64, m64*	ZO	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS *m8*	ZO	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS *m8*	ZO	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS *m16*	ZO	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m32*	ZO	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS *m64*	ZO	Valid	N.E.	Find RAX, starting at [RDI].

NOTES:

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

Instruction Operand Encoding

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

Description

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The F3H prefix is defined for the following instructions and undefined for the rest:

- F3H as REP/REPE/REPZ for string and input/output instruction.
- F3H is a mandatory prefix for POPCNT, LZCNT, and ADOX.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-16.

Table 4-16. Repeat Prefixes

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

NOTES:

* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

REP INS 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 AddressSize = 16
  THEN
    Use CX for CountReg;
    Implicit Source/Dest operand for memory use of SI/DI;
  ELSE IF AddressSize = 64
    THEN Use RCX for CountReg;
    Implicit Source/Dest operand for memory use of RSI/RDI;
  ELSE
    Use ECX for CountReg;
    Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
   DO
      Service pending interrupts (if any);
      Execute associated string instruction;
      CountReg ← (CountReg – 1);
      IF CountReg = 0
          THEN exit WHILE loop; FI;
      IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
          THEN exit WHILE loop; FI;
   OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions

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

RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 *iw*	RET *imm16*	I	Valid	Valid	Near return to calling procedure and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	I	Valid	Valid	Far return to calling procedure and pop *imm16* bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm16	NA	NA	NA

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

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

Unlike near indirect CALL and near indirect JMP, the processor will not speculatively execute the next sequential instruction after a near RET unless that instruction is also the target of a jump or is a target in a branch predictor.

Operation

```
(* Near return *)
IF instruction = near return
    THEN;
        IF OperandSize = 32
            THEN
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
            ELSE
                IF OperandSize = 64
                    THEN
                        IF top 8 bytes of stack not within stack limits
                            THEN #SS(0); FI;
                        RIP ← Pop();
                    ELSE (* OperandSize = 16 *)
                        IF top 2 bytes of stack not within stack limits
                            THEN #SS(0); FI;
                        tempEIP ← Pop();
                        tempEIP ← tempEIP AND 0000FFFFH;
                        IF tempEIP not within code segment limits
                            THEN #GP(0); FI;
                        EIP ← tempEIP;
                FI;
        FI;

    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE
                    IF StackAddressSize = 64
                        THEN
                            RSP ← RSP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
    FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
```

```
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP ← tempEIP;
                CS ← Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP ← SP + (SRC AND FFFFH);
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
        FI;
    IF return code segment selector is NULL
        THEN #GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN #GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming and return code
    segment DPL ≠ return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;
```

```
RETURN-TO-SAME-PRIVILEGE-LEVEL:
   IF the return instruction pointer is not within the return code segment limit
         THEN #GP(0); FI;
   IF OperandSize = 32
         THEN
               EIP ← Pop();
               CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
         ELSE (* OperandSize = 16 *)
               EIP ← Pop();
               EIP ← EIP AND 0000FFFFH;
               CS ← Pop(); (* 16-bit pop *)
   FI;
   IF instruction has immediate operand
         THEN (* Release parameters from stack *)
               IF StackAddressSize = 32
                     THEN
                           ESP ← ESP + SRC;
                     ELSE (* StackAddressSize = 16 *)
                           SP ← SP + SRC;
               FI;
   FI;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
   IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
   or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
               THEN #SS(0); FI;
   Read return segment selector;
   IF stack segment selector is NULL
         THEN #GP(0); FI;
   IF return stack segment selector index is not within its descriptor table limits
         THEN #GP(selector); FI;
   Read segment descriptor pointed to by return segment selector;
   IF stack segment selector RPL ≠ RPL of the return code segment selector
   or stack segment is not a writable data segment
   or stack segment descriptor DPL ≠ RPL of the return code segment selector
               THEN #GP(selector); FI;
   IF stack segment not present
         THEN #SS(StackSegmentSelector); FI;
   IF the return instruction pointer is not within the return code segment limit
         THEN #GP(0); FI;
   CPL ← ReturnCodeSegmentSelector(RPL);
   IF OperandSize = 32
         THEN
               EIP ← Pop();
               CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
               CS(RPL) ← CPL;
               IF instruction has immediate operand
                     THEN (* Release parameters from called procedure’s stack *)
                           IF StackAddressSize = 32
                                 THEN
                                       ESP ← ESP + SRC;
                                 ELSE (* StackAddressSize = 16 *)
                                       SP ← SP + SRC;
                           FI;
```

```
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
    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;

    IF instruction has immediate operand
        THEN (* Release parameters from calling procedure's stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
    FI;

(* IA-32e Mode *)
    IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
        THEN
            IF OperandSize = 32
                THEN
                    IF second doubleword on stack is not within stack limits
                        THEN #SS(0); FI;
                    IF first or second doubleword on stack is not in canonical space
                        THEN #SS(0); FI;
                ELSE
```

```
                    IF OperandSize = 16
                        THEN
                            IF second word on stack is not within stack limits
                                THEN #SS(0); FI;
                            IF first or second word on stack is not in canonical space
                                THEN #SS(0); FI;
                        ELSE (* OperandSize = 64 *)
                            IF first or second quadword on stack is not in canonical space
                                THEN #SS(0); FI;
                    FI
            FI;
        IF return code segment selector is NULL
            THEN GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN GP(selector); FI;
        IF return code segment selector addresses descriptor in non-canonical space
            THEN GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment
            THEN #GP(selector); FI;
        IF return code segment descriptor has L-bit = 1 and D-bit = 1
            THEN #GP(selector); FI;
        IF return code segment selector RPL < CPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is conforming
        and return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is non-conforming
        and return code segment DPL ≠ return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is not present
            THEN #NP(selector); FI:
        IF return code segment selector RPL > CPL
            THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
        FI;
    FI;

IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
```

```
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP ← SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP ← RSP + SRC;
                FI;
        FI;
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
FI;
IF return stack segment descriptor is not within descriptor table limits
        THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
        THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI:
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
```

```
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE
                        IF StackAddressSize = 16
                            THEN
                                SP ← SP + SRC;
                            ELSE (* StackAddressSize = 64 *)
                                RSP ← RSP + SRC;
                        FI;
                FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        IF StackAddressSize = 32
                            THEN
                                ESP ← ESP + SRC;
                            ELSE
                                IF StackAddressSize = 16
                                    THEN
                                        SP ← SP + SRC;
                                    ELSE (* StackAddressSize = 64 *)
                                        RSP ← RSP + SRC;
                                FI;
                        FI;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                ESP ← tempESP;
                SS ← tempSS;
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        RSP ← RSP + SRC;
                FI;
                tempESP ← Pop();
```

```
                tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                ESP ← tempESP;
                SS ← tempSS;
        FI;
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP ← SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP ← RSP + SRC;
                FI;
        FI;
FI;
```

Flags Affected

None.

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 the RPL of the return code segment selector is less then the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
	If the return instruction pointer is not within the return code segment limit.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return code segment selector is NULL.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

RORX — Rotate Right Logical Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, *r/m32, imm8*	RMI	V/V	BMI2	Rotate 32-bit *r/m32* right *imm8* times without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, *r/m64, imm8*	RMI	V/N.E.	BMI2	Rotate 64-bit *r/m64* right *imm8* times without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

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

```
IF (OperandSize = 32)
    y ← imm8 AND 1FH;
    DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64 )
    y ← imm8 AND 3FH;
    DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 13.

ROUNDPD — Round Packed Double Precision Floating-Point Values

Opcode*/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 3A 09 /r ib ROUNDPD *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Round packed double precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Round packed double-precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX	Round packed double-precision floating-point values in *ymm2/m256* and place the result in *ymm1*. The rounding mode is determined by *imm8*.

Instruction Operand Encoding

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

Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a double-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-17 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 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 source operand second source operand or a 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 source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

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

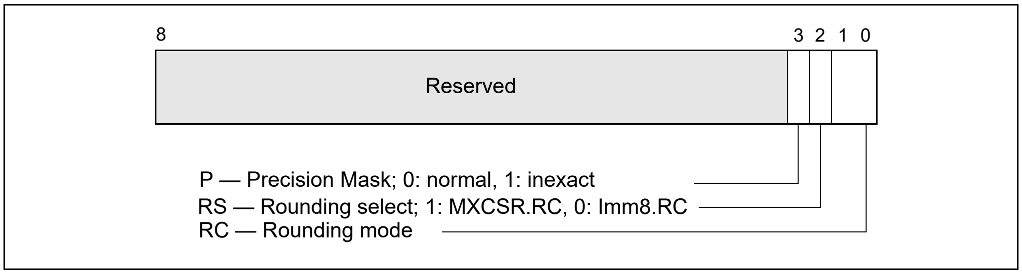

Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-17. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI
```

ROUNDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)
```

VROUNDPD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0
```

VROUNDPD (VEX.256 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[191:128] ← RoundToInteger(SRC[191:128]], ROUND_CONTROL)
DEST[255:192] ← RoundToInteger(SRC[255:192] ], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_pd(__m128d s1, int iRoundMode);

__m128 _mm_floor_pd(__m128d s1);

__m128 _mm_ceil_pd(__m128d s1)

__m256 _mm256_round_pd(__m256d s1, int iRoundMode);

__m256 _mm256_floor_pd(__m256d s1);

__m256 _mm256_ceil_pd(__m256d s1)

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv ≠ 1111B.

ROUNDPS — Round Packed Single Precision Floating-Point Values

Opcode*/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 3A 08 /r ib ROUNDPS *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Round packed single precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Round packed single-precision floating-point values in *xmm2/m128* and place the result in xmm1. The rounding mode is determined by *imm8*.
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX	Round packed single-precision floating-point values in *ymm2/m256* and place the result in ymm1. The rounding mode is determined by *imm8*.

Instruction Operand Encoding

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

Description

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-17 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 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 source operand second source operand or a 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 source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

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

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;
```

ROUNDPS(128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)
```

VROUNDPS (VEX.128 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0
```

VROUNDPS (VEX.256 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[159:128] ← RoundToInteger(SRC[159:128]], ROUND_CONTROL)
DEST[191:160] ← RoundToInteger(SRC[191:160]], ROUND_CONTROL)
DEST[223:192] ← RoundToInteger(SRC[223:192] ], ROUND_CONTROL)
DEST[255:224] ← RoundToInteger(SRC[255:224] ], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_ps(__m128 s1, int iRoundMode);

__m128 _mm_floor_ps(__m128 s1);

__m128 _mm_ceil_ps(__m128 s1)

__m256 _mm256_round_ps(__m256 s1, int iRoundMode);

__m256 _mm256_floor_ps(__m256 s1);

__m256 _mm256_ceil_ps(__m256 s1)

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv ≠ 1111B.

ROUNDSD — Round Scalar Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0B /r ib ROUNDSD *xmm1, xmm2/m64, imm8*	RMI	V/V	SSE4_1	Round the low packed double precision floating-point value in *xmm2/m64* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.LIG.66.0F3A.WIG 0B /r ib VROUNDSD *xmm1, xmm2, xmm3/m64, imm8*	RVMI	V/V	AVX	Round the low packed double precision floating-point value in *xmm3/m64* and place the result in *xmm1*. The rounding mode is determined by *imm8*. Upper packed double precision floating-point value (bits[127:64]) from *xmm2* is copied to *xmm1*[127:64].

Instruction Operand Encoding

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

Description

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-17 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

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

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

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

ROUNDSD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[MAXVL-1:64] (Unmodified)
```

VROUNDSD (VEX.128 encoded version)

DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSD: __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
__m128d mm_floor_sd(__m128d dst, __m128d s1);
__m128d mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Exceptions Type 3.

ROUNDSS — Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A /r ib ROUNDSS *xmm1, xmm2/m32, imm8*	RMI	V/V	SSE4_1	Round the low packed single precision floating-point value in *xmm2/m32* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.LIG.66.0F3A.WIG 0A /r ib VROUNDSS *xmm1, xmm2, xmm3/m32, imm8*	RVMI	V/V	AVX	Round the low packed single precision floating-point value in *xmm3/m32* and place the result in *xmm1*. The rounding mode is determined by *imm8*. Also, upper packed single precision floating-point values (bits[127:32]) from *xmm2* are copied to *xmm1*[127:32].

Instruction Operand Encoding

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

Description

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer value and returns the result as a single-precision floating-point value in the lowest position. The upper three single-precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-17 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

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: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN      // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE      // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

ROUNDSS (128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[MAXVL-1:32] (Unmodified)
```

VROUNDSS (VEX.128 encoded version)

DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSS: __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
__m128 mm_floor_ss(__m128 dst, __m128 s1);
__m128 mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Exceptions Type 3.

RSM—Resume from System Management Mode

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AA	RSM	ZO	Valid	Valid	Resume operation of interrupted program.

Instruction Operand Encoding

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

Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 34, "System Management Mode," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about SMM and the behavior of the RSM instruction.

Operation

```
ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI
```

Flags Affected

All.

Protected Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.
	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.

RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 52 /r RSQRTPS *xmm1, xmm2/m128*	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.0F.WIG 52 /r VRSQRTPS *xmm1, xmm2/m128*	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in *xmm2/mem* and stores the results in *xmm1*.
VEX.256.0F.WIG 52 /r VRSQRTPS *ymm1, ymm2/m256*	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in *ymm2/mem* and stores the results in *ymm1*.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

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.

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

Operation

RSQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] (Unmodified)
```

VRSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] ← 0
```

VRSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC2[255:224]))
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS: __m128 _mm_rsqrt_ps(__m128 a)

RSQRTPS: __m256 _mm256_rsqrt_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
F3 0F 52 /*r* RSQRTSS *xmm1*, *xmm2/m32*	RM	V/V	SSE	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*.
VEX.LIG.F3.0F.WIG 52 /r VRSQRTSS *xmm1, xmm2, xmm3/m32*	RVM	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in *xmm3/m32* and stores the results in *xmm1*. Also, upper single precision floating-point values (bits[127:32]) from *xmm2* are copied to *xmm1*[127:32].

Instruction Operand Encoding

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

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

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: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RSQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[MAXVL-1:32] (Unmodified)
```

VRSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS: __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

SAHF—Store AH into Flags

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9E	SAHF	ZO	Invalid*	Valid	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.

NOTES:

* Valid in specific steppings. See Description section.

Instruction Operand Encoding

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

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation

```
IF IA-64 Mode
    THEN
        IF CPUID.80000001H.ECX[0] = 1;
            THEN
                RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
            ELSE
                #UD;
        FI
    ELSE
        EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
FI;
```

Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Compatibility Mode Exceptions

None.

64-Bit Mode Exceptions

#UD	If CPUID.80000001H.ECX[0] = 0.
	If the LOCK prefix is used.

SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL *r/m8*, 1	M1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SAL *r/m8***, 1	M1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SAL *r/m8*, CL	MC	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SAL *r/m8***, CL	MC	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SAL *r/m8, imm8*	MI	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SAL *r/m8**, imm8*	MI	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SAL *r/m16*, 1	M1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SAL *r/m16*, CL	MC	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SAL *r/m16, imm8*	MI	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SAL *r/m32*, 1	M1	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SAL *r/m64*, 1	M1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SAL *r/m32*, CL	MC	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SAL *r/m64*, CL	MC	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SAL *r/m32, imm8*	MI	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SAL *r/m64, imm8*	MI	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /7	SAR *r/m8*, 1	M1	Valid	Valid	Signed divide* *r/m8* by 2, once.
REX + D0 /7	SAR *r/m8***, 1	M1	Valid	N.E.	Signed divide* *r/m8* by 2, once.
D2 /7	SAR *r/m8*, CL	MC	Valid	Valid	Signed divide* *r/m8* by 2, CL times.
REX + D2 /7	SAR *r/m8***, CL	MC	Valid	N.E.	Signed divide* *r/m8* by 2, CL times.
C0 /7 *ib*	SAR *r/m8, imm8*	MI	Valid	Valid	Signed divide* *r/m8* by 2, *imm8* time.
REX + C0 /7 *ib*	SAR *r/m8**, imm8*	MI	Valid	N.E.	Signed divide* *r/m8* by 2, *imm8* times.
D1 /7	SAR *r/m16*,1	M1	Valid	Valid	Signed divide* *r/m16* by 2, once.
D3 /7	SAR *r/m16*, CL	MC	Valid	Valid	Signed divide* *r/m16* by 2, CL times.
C1 /7 *ib*	SAR *r/m16, imm8*	MI	Valid	Valid	Signed divide* *r/m16* by 2, *imm8* times.
D1 /7	SAR *r/m32*, 1	M1	Valid	Valid	Signed divide* *r/m32* by 2, once.
REX.W + D1 /7	SAR *r/m64*, 1	M1	Valid	N.E.	Signed divide* *r/m64* by 2, once.
D3 /7	SAR *r/m32*, CL	MC	Valid	Valid	Signed divide* *r/m32* by 2, CL times.
REX.W + D3 /7	SAR *r/m64*, CL	MC	Valid	N.E.	Signed divide* *r/m64* by 2, CL times.
C1 /7 *ib*	SAR *r/m32, imm8*	MI	Valid	Valid	Signed divide* *r/m32* by 2, *imm8* times.
REX.W + C1 /7 *ib*	SAR *r/m64, imm8*	MI	Valid	N.E.	Signed divide* *r/m64* by 2, *imm8* times
D0 /4	SHL *r/m8*, 1	M1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SHL *r/m8***, 1	M1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SHL *r/m8*, CL	MC	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SHL *r/m8***, CL	MC	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SHL *r/m8, imm8*	MI	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SHL *r/m8**, imm8*	MI	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SHL *r/m16*,1	M1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SHL *r/m16*, CL	MC	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SHL *r/m16, imm8*	MI	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SHL *r/m32*,1	M1	Valid	Valid	Multiply *r/m32* by 2, once.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D1 /4	SHL *r/m64*,1	M1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SHL *r/m32*, CL	MC	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SHL *r/m64*, CL	MC	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SHL *r/m32, imm8*	MI	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SHL *r/m64, imm8*	MI	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /5	SHR *r/m8*,1	M1	Valid	Valid	Unsigned divide *r/m8* by 2, once.
REX + D0 /5	SHR *r/m8***, 1	M1	Valid	N.E.	Unsigned divide *r/m8* by 2, once.
D2 /5	SHR *r/m8*, CL	MC	Valid	Valid	Unsigned divide *r/m8* by 2, CL times.
REX + D2 /5	SHR *r/m8***, CL	MC	Valid	N.E.	Unsigned divide *r/m8* by 2, CL times.
C0 /5 *ib*	SHR *r/m8, imm8*	MI	Valid	Valid	Unsigned divide *r/m8* by 2, *imm8* times.
REX + C0 /5 *ib*	SHR *r/m8**, imm8*	MI	Valid	N.E.	Unsigned divide *r/m8* by 2, *imm8* times.
D1 /5	SHR *r/m16*, 1	M1	Valid	Valid	Unsigned divide *r/m16* by 2, once.
D3 /5	SHR *r/m16*, CL	MC	Valid	Valid	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16, imm8*	MI	Valid	Valid	Unsigned divide *r/m16* by 2, *imm8* times.
D1 /5	SHR *r/m32*, 1	M1	Valid	Valid	Unsigned divide *r/m32* by 2, once.
REX.W + D1 /5	SHR *r/m64*, 1	M1	Valid	N.E.	Unsigned divide *r/m64* by 2, once.
D3 /5	SHR *r/m32*, CL	MC	Valid	Valid	Unsigned divide *r/m32* by 2, CL times.
REX.W + D3 /5	SHR *r/m64*, CL	MC	Valid	N.E.	Unsigned divide *r/m64* by 2, CL times.
C1 /5 *ib*	SHR *r/m32, imm8*	MI	Valid	Valid	Unsigned divide *r/m32* by 2, *imm8* times.
REX.W + C1 /5 *ib*	SHR *r/m64, imm8*	MI	Valid	N.E.	Unsigned divide *r/m64* by 2, *imm8* times.

NOTES:

* Not the same form of division as IDIV; rounding is toward negative infinity.

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

***See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (r, w)	1	NA	NA
MC	ModRM:r/m (r, w)	CL	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction's default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
IF 64-Bit Mode and using REX.W
    THEN
        countMASK ← 3FH;
    ELSE
        countMASK ← 1FH;
FI

tempCOUNT ← (COUNT AND countMASK);
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
    IF instruction is SAL or SHL
        THEN
            CF ← MSB(DEST);
        ELSE (* Instruction is SAR or SHR *)
            CF ← LSB(DEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            DEST ← DEST ∗ 2;
        ELSE
            IF instruction is SAR
```

```
                THEN
                    DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
                ELSE (* Instruction is SHR *)
                    DEST ← DEST / 2 ; (* Unsigned divide *)
            FI;
    FI;
    tempCOUNT ← tempCOUNT – 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
        FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF ← undefined;
    FI;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

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

SARX/SHLX/SHRX — Shift Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F3.0F38.W0 F7 /r SARX *r32a, r/m32, r32b*	RMV	V/V	BMI2	Shift *r/m32* arithmetically right with count specified in *r32b.*
VEX.LZ.66.0F38.W0 F7 /r SHLX *r32a, r/m32, r32b*	RMV	V/V	BMI2	Shift *r/m32* logically left with count specified in *r32b*.
VEX.LZ.F2.0F38.W0 F7 /r SHRX *r32a, r/m32, r32b*	RMV	V/V	BMI2	Shift *r/m32* logically right with count specified in *r32b*.
VEX.LZ.F3.0F38.W1 F7 /r SARX *r64a, r/m64, r64b*	RMV	V/N.E.	BMI2	Shift *r/m64* arithmetically right with count specified in *r64b*.
VEX.LZ.66.0F38.W1 F7 /r SHLX *r64a, r/m64, r64b*	RMV	V/N.E.	BMI2	Shift *r/m64* logically left with count specified in *r64b*.
VEX.LZ.F2.0F38.W1 F7 /r SHRX *r64a, r/m64, r64b*	RMV	V/N.E.	BMI2	Shift *r/m64* logically right with count specified in *r64b*.

Instruction Operand Encoding

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

Description

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

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.

If the value specified in the first source operand exceeds OperandSize -1, the COUNT value is masked.

SARX,SHRX, and SHLX instructions do not update flags.

Operation

```
TEMP ← SRC1;
IF VEX.W1 and CS.L = 1
THEN
    countMASK ←3FH;
ELSE
    countMASK ←1FH;
FI
COUNT ← (SRC2 AND countMASK)

DEST[OperandSize -1] = TEMP[OperandSize -1];
DO WHILE (COUNT ≠ 0)
    IF instruction is SHLX
        THEN
            DEST[] ← DEST *2;
```

```
        ELSE IF instruction is SHRX
            THEN
                DEST[] ← DEST /2; //unsigned divide
        ELSE            // SARX
                DEST[] ← DEST /2; // signed divide, round toward negative infinity
    FI;
    COUNT ← COUNT - 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 13.

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
1C *ib*	SBB AL, *imm8*	I	Valid	Valid	Subtract with borrow *imm8* from AL.
1D *iw*	SBB AX, *imm16*	I	Valid	Valid	Subtract with borrow *imm16* from AX.
1D *id*	SBB EAX, *imm32*	I	Valid	Valid	Subtract with borrow *imm32* from EAX.
REX.W + 1D *id*	SBB RAX, *imm32*	I	Valid	N.E.	Subtract with borrow sign-extended *imm.32 to 64-bits* from RAX.
80 /3 *ib*	SBB *r/m8, imm8*	MI	Valid	Valid	Subtract with borrow *imm8* from *r/m8.*
REX + 80 /3 *ib*	SBB *r/m8*, imm8*	MI	Valid	N.E.	Subtract with borrow *imm8* from *r/m8.*
81 /3 *iw*	SBB *r/m16, imm16*	MI	Valid	Valid	Subtract with borrow *imm16* from *r/m16.*
81 /3 *id*	SBB *r/m32, imm32*	MI	Valid	Valid	Subtract with borrow *imm32* from *r/m32.*
REX.W + 81 /3 *id*	SBB *r/m64, imm32*	MI	Valid	N.E.	Subtract with borrow sign-extended *imm32 to 64-bits* from *r/m64.*
83 /3 *ib*	SBB *r/m16, imm8*	MI	Valid	Valid	Subtract with borrow sign-extended *imm8* from *r/m16.*
83 /3 *ib*	SBB *r/m32, imm8*	MI	Valid	Valid	Subtract with borrow sign-extended *imm8* from *r/m32.*
REX.W + 83 /3 *ib*	SBB *r/m64, imm8*	MI	Valid	N.E.	Subtract with borrow sign-extended *imm8* from *r/m64.*
18 /*r*	SBB *r/m8, r8*	MR	Valid	Valid	Subtract with borrow *r8* from *r/m8.*
REX + 18 /*r*	SBB *r/m8*, r8*	MR	Valid	N.E.	Subtract with borrow *r8* from *r/m8.*
19 /*r*	SBB *r/m16, r16*	MR	Valid	Valid	Subtract with borrow *r16* from *r/m16.*
19 /*r*	SBB *r/m32, r32*	MR	Valid	Valid	Subtract with borrow *r32* from *r/m32.*
REX.W + 19 /*r*	SBB *r/m64, r64*	MR	Valid	N.E.	Subtract with borrow *r64* from *r/m64.*
1A /*r*	SBB *r8, r/m8*	RM	Valid	Valid	Subtract with borrow *r/m8* from *r8.*
REX + 1A /*r*	SBB *r8*, r/m8**	RM	Valid	N.E.	Subtract with borrow *r/m8* from *r8.*
1B /*r*	SBB *r16, r/m16*	RM	Valid	Valid	Subtract with borrow *r/m16* from *r16.*
1B /*r*	SBB *r32, r/m32*	RM	Valid	Valid	Subtract with borrow *r/m32* from *r32.*
REX.W + 1B /*r*	SBB *r64, r/m64*	RM	Valid	N.E.	Subtract with borrow *r/m64* from *r64.*

NOTES:

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

Instruction Operand Encoding

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

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

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

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

Operation

```
DEST ← (DEST – (SRC + CF));
```

Intel C/C++ Compiler Intrinsic Equivalent

SBB: extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);

SBB: extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);

SBB: extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *diff_out);

SBB: extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);

Flags Affected

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

Protected Mode Exceptions

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

SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AE	SCAS *m8*	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.*
AF	SCAS *m16*	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.*
AF	SCAS *m32*	ZO	Valid	Valid	Compare EAX with doubleword at ES(E)DI or RDI then set status flags.*
REX.W + AF	SCAS *m64*	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.*
AF	SCASW	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.*
AF	SCASD	ZO	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCASQ	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

NOTES:

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

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

Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of status flags. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction's default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
Non-64-bit Mode:

IF (Byte comparison)
    THEN
        temp ← AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (E)DI ← (E)DI + 1;
                ELSE (E)DI ← (E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp ← AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI ← (E)DI + 2;
                ELSE (E)DI ← (E)DI – 2; FI;
        FI;
    ELSE IF (Doubleword comparison)
        THEN
            temp ← EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI – 4; FI;
        FI;
FI;

64-bit Mode:

IF (Byte cmparison)
    THEN
        temp ← AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 1;
                ELSE (R|E)DI ← (R|E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp ← AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 2;
                ELSE (R|E)DI ← (R|E)DI – 2; FI;
        FI;
```

```
    ELSE IF (Doubleword comparison)
        THEN
            temp ← EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 4;
                ELSE (R|E)DI ← (R|E)DI – 4; FI;
        FI;
    ELSE IF (Quadword comparison using REX.W )
        THEN
            temp ← RAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 8;
                ELSE (R|E)DI ← (R|E)DI – 8;
            FI;
    FI;
F
```

Flags Affected

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

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a NULL segment selector.
	If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#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

#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.

SET*cc*—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA *r/m8*	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 97	SETA *r/m8**	M	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE *r/m8*	M	Valid	Valid	Set byte if above or equal (CF=0).
REX + 0F 93	SETAE *r/m8**	M	Valid	N.E.	Set byte if above or equal (CF=0).
0F 92	SETB *r/m8*	M	Valid	Valid	Set byte if below (CF=1).
REX + 0F 92	SETB *r/m8**	M	Valid	N.E.	Set byte if below (CF=1).
0F 96	SETBE *r/m8*	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + 0F 96	SETBE *r/m8**	M	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC *r/m8*	M	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC *r/m8**	M	Valid	N.E.	Set byte if carry (CF=1).
0F 94	SETE *r/m8*	M	Valid	Valid	Set byte if equal (ZF=1).
REX + 0F 94	SETE *r/m8**	M	Valid	N.E.	Set byte if equal (ZF=1).
0F 9F	SETG *r/m8*	M	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
REX + 0F 9F	SETG *r/m8**	M	Valid	N.E.	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE *r/m8*	M	Valid	Valid	Set byte if greater or equal (SF=OF).
REX + 0F 9D	SETGE *r/m8**	M	Valid	N.E.	Set byte if greater or equal (SF=OF).
0F 9C	SETL *r/m8*	M	Valid	Valid	Set byte if less (SF≠ OF).
REX + 0F 9C	SETL *r/m8**	M	Valid	N.E.	Set byte if less (SF≠ OF).
0F 9E	SETLE *r/m8*	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
REX + 0F 9E	SETLE *r/m8**	M	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA *r/m8*	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + 0F 96	SETNA *r/m8**	M	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE *r/m8*	M	Valid	Valid	Set byte if not above or equal (CF=1).
REX + 0F 92	SETNAE *r/m8**	M	Valid	N.E.	Set byte if not above or equal (CF=1).
0F 93	SETNB *r/m8*	M	Valid	Valid	Set byte if not below (CF=0).
REX + 0F 93	SETNB *r/m8**	M	Valid	N.E.	Set byte if not below (CF=0).
0F 97	SETNBE *r/m8*	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
REX + 0F 97	SETNBE *r/m8**	M	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC *r/m8*	M	Valid	Valid	Set byte if not carry (CF=0).
REX + 0F 93	SETNC *r/m8**	M	Valid	N.E.	Set byte if not carry (CF=0).
0F 95	SETNE *r/m8*	M	Valid	Valid	Set byte if not equal (ZF=0).
REX + 0F 95	SETNE *r/m8**	M	Valid	N.E.	Set byte if not equal (ZF=0).
0F 9E	SETNG *r/m8*	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF)
REX + 0F 9E	SETNG *r/m8**	M	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ OF).
0F 9C	SETNGE *r/m8*	M	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
REX + 0F 9C	SETNGE *r/m8**	M	Valid	N.E.	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL *r/m8*	M	Valid	Valid	Set byte if not less (SF=OF).
REX + 0F 9D	SETNL *r/m8**	M	Valid	N.E.	Set byte if not less (SF=OF).
0F 9F	SETNLE *r/m8*	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + 0F 9F	SETNLE *r/m8**	M	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO *r/m8*	M	Valid	Valid	Set byte if not overflow (OF=0).
REX + 0F 91	SETNO *r/m8**	M	Valid	N.E.	Set byte if not overflow (OF=0).
0F 9B	SETNP *r/m8*	M	Valid	Valid	Set byte if not parity (PF=0).
REX + 0F 9B	SETNP *r/m8**	M	Valid	N.E.	Set byte if not parity (PF=0).
0F 99	SETNS *r/m8*	M	Valid	Valid	Set byte if not sign (SF=0).
REX + 0F 99	SETNS *r/m8**	M	Valid	N.E.	Set byte if not sign (SF=0).
0F 95	SETNZ *r/m8*	M	Valid	Valid	Set byte if not zero (ZF=0).
REX + 0F 95	SETNZ *r/m8**	M	Valid	N.E.	Set byte if not zero (ZF=0).
0F 90	SETO *r/m8*	M	Valid	Valid	Set byte if overflow (OF=1)
REX + 0F 90	SETO *r/m8**	M	Valid	N.E.	Set byte if overflow (OF=1).
0F 9A	SETP *r/m8*	M	Valid	Valid	Set byte if parity (PF=1).
REX + 0F 9A	SETP *r/m8**	M	Valid	N.E.	Set byte if parity (PF=1).
0F 9A	SETPE *r/m8*	M	Valid	Valid	Set byte if parity even (PF=1).
REX + 0F 9A	SETPE *r/m8**	M	Valid	N.E.	Set byte if parity even (PF=1).
0F 9B	SETPO *r/m8*	M	Valid	Valid	Set byte if parity odd (PF=0).
REX + 0F 9B	SETPO *r/m8**	M	Valid	N.E.	Set byte if parity odd (PF=0).
0F 98	SETS *r/m8*	M	Valid	Valid	Set byte if sign (SF=1).
REX + 0F 98	SETS *r/m8**	M	Valid	N.E.	Set byte if sign (SF=1).
0F 94	SETZ *r/m8*	M	Valid	Valid	Set byte if zero (ZF=1).
REX + 0F 94	SETZ *r/m8**	M	Valid	N.E.	Set byte if zero (ZF=1).

NOTES:

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

Instruction Operand Encoding

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

Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

The reg field of the ModR/M byte is not used for the SETCC instruction and those opcode bits are ignored by the processor.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

Operation

```
IF condition
    THEN DEST ← 1;
    ELSE DEST ← 0;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

SFENCE—Store Fence

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F8	SFENCE	ZO	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

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

Description

Orders processor execution relative to all memory stores prior to the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible. The SFENCE instruction is ordered with respect to memory stores, other SFENCE instructions, MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to memory loads or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

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

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

Operation

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_sfence(void)
```

Exceptions (All Operating Modes)

#UD	If CPUID.01H:EDX.SSE[bit 25] = 0.
	If the LOCK prefix is used.

SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT *m*	M	Valid	Valid	Store GDTR to *m.*

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 (w)	NA	NA	NA

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 or 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SGDT
        IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
            THEN
                DEST[0:15] ← GDTR(Limit);
                DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)
                FI;
            ELSE (* 64-bit Mode *)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#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.
	If CR4.UMIP = 1 and CPL > 0.
#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 CPL = 3.

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(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SHA1RNDS4—Perform Four Rounds of SHA1 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A CC /r ib SHA1RNDS4 xmm1, xmm2/m128, imm8	RMI	V/V	SHA	Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8

Description

The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

Operation

SHA1RNDS4

The function f() and Constant K are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN f() ← f0(), K ← K0;
ELSE IF ( imm8[1:0] = 1 )
    THEN f() ← f1(), K ← K1;
ELSE IF ( imm8[1:0] = 2 )
    THEN f() ← f2(), K ← K2;
ELSE IF ( imm8[1:0] = 3 )
    THEN f() ← f3(), K ← K3;
FI;

A ← SRC1[127:96];
B ← SRC1[95:64];
C ← SRC1[63:32];
D ← SRC1[31:0];
W0E ← SRC2[127:96];
W1 ← SRC2[95:64];
W2 ← SRC2[63:32];
W3 ← SRC2[31:0];

Round i = 0 operation:
A_1 ← f (B, C, D) + (A ROL 5) +W0E +K;
B_1 ← A;
C_1 ← B ROL 30;
D_1 ← C;
E_1 ← D;

FOR i = 1 to 3
    A_(i +1) ← f (B_i, C_i, D_i) + (A_i ROL 5) +Wi+ E_i +K;
    B_(i +1) ← A_i;
```

```
    C_(i +1) ← B_i ROL 30;
    D_(i +1) ← C_i;
    E_(i +1) ← D_i;
ENDFOR

DEST[127:96] ← A_4;
DEST[95:64] ← B_4;
DEST[63:32] ← C_4;
DEST[31:0] ← D_4;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1RNDS4: __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C8 /r SHA1NEXTE xmm1, xmm2/m128	RM	V/V	SHA	Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.

Instruction Operand Encoding

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

Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

Operation

SHA1NEXTE

```
TMP ← (SRC1[127:96] ROL 30);

DEST[127:96] ← SRC2[127:96] + TMP;
DEST[95:64] ← SRC2[95:64];
DEST[63:32] ← SRC2[63:32];
DEST[31:0] ← SRC2[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE: __m128i _mm_sha1nexte_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

Operation

SHA1MSG1

```
W0 ← SRC1[127:96] ;
W1 ← SRC1[95:64] ;
W2 ← SRC1[63: 32] ;
W3 ← SRC1[31: 0] ;
W4 ← SRC2[127:96] ;
W5 ← SRC2[95:64] ;

DEST[127:96] ← W2 XOR W0;
DEST[95:64] ← W3 XOR W1;
DEST[63:32] ← W4 XOR W2;
DEST[31:0] ← W5 XOR W3;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG1: __m128i _mm_sha1msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CA /r SHA1MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

Operation

SHA1MSG2

```
W13 ← SRC2[95:64] ;
W14 ← SRC2[63: 32] ;
W15 ← SRC2[31: 0] ;
W16 ← (SRC1[127:96] XOR W13 ) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63: 32] XOR W15) ROL 1;
W19 ← (SRC1[31: 0] XOR W16) ROL 1;

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256RNDS2—Perform Two Rounds of SHA256 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CB /r SHA256RNDS2 xmm1, xmm2/m128, <XMM0>	RMO	V/V	SHA	Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Implicit XMM0 (r)

Description

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

Operation

SHA256RNDS2

A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK_0 ← XMM0[31: 0];
WK_1 ← XMM0[63: 32];

FOR i = 0 to 1
    A_(i +1) ← Ch (E_i, F_i, G_i) +Σ_1(E_i) +WK_i+ H_i + Maj(A_i , B_i, C_i) +Σ_0(A_i);
    B_(i +1) ← A_i;
    C_(i +1) ← B_i ;
    D_(i +1) ← C_i;
    E_(i +1) ← Ch (E_i, F_i, G_i) +Σ_1(E_i) +WK_i+ H_i + D_i;
    F_(i +1) ← E_i ;
    G_(i +1) ← F_i;
    H_(i +1) ← G_i;
ENDFOR

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256RNDS2: __m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CC /r SHA256MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

Operation

SHA256MSG1

W4 ← SRC2[31: 0] ;
W3 ← SRC1[127:96] ;
W2 ← SRC1[95:64] ;
W1 ← SRC1[63: 32] ;
W0 ← SRC1[31: 0] ;

DEST[127:96] ← W3 + σ_0(W4);
DEST[95:64] ← W2 + σ_0(W3);
DEST[63:32] ← W1 + σ_0(W2);
DEST[31:0] ← W0 + σ_0(W1);

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG1: __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

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

Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

Operation

SHA256MSG2

W14 ← SRC2[95:64] ;
W15 ← SRC2[127:96] ;
W16 ← SRC1[31: 0] + σ_1(W14) ;
W17 ← SRC1[63: 32] + σ_1(W15) ;
W18 ← SRC1[95: 64] + σ_1(W16) ;
W19 ← SRC1[127: 96] + σ_1(W17) ;

DEST[127:96] ← W19 ;
DEST[95:64] ← W18 ;
DEST[63:32] ← W17 ;
DEST[31:0] ← W16;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 : __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHLD—Double Precision Shift Left

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A4 /r ib	SHLD *r/m16, r16, imm8*	MRI	Valid	Valid	Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right.
0F A5 /r	SHLD *r/m16, r16*, CL	MRC	Valid	Valid	Shift *r/m16* to left CL places while shifting bits from *r16* in from the right.
0F A4 /r ib	SHLD *r/m32, r32, imm8*	MRI	Valid	Valid	Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right.
REX.W + 0F A4 /r ib	SHLD *r/m64, r64, imm8*	MRI	Valid	N.E.	Shift *r/m64* to left *imm8* places while shifting bits from *r64* in from the right.
0F A5 /r	SHLD *r/m32, r32*, CL	MRC	Valid	Valid	Shift *r/m32* to left CL places while shifting bits from *r32* in from the right.
REX.W + 0F A5 /r	SHLD *r/m64, r64*, CL	MRC	Valid	N.E.	Shift *r/m64* to left CL places while shifting bits from *r64* in from the right.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
```

```
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, SIZE – COUNT];
                (* Last bit shifted out on exit *)
                FOR i ← SIZE – 1 DOWN TO COUNT
                    DO
                        Bit(DEST, i) ← Bit(DEST, i – COUNT);
                    OD;
                FOR i ← COUNT – 1 DOWN TO 0
                    DO
                        BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
                    OD;
        FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions

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

SHRD—Double Precision Shift Right

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AC /r ib	SHRD *r/m16, r16, imm8*	MRI	Valid	Valid	Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left.
0F AD /r	SHRD *r/m16, r16,* CL	MRC	Valid	Valid	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left.
0F AC /r ib	SHRD *r/m32, r32, imm8*	MRI	Valid	Valid	Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left.
REX.W + 0F AC /r ib	SHRD *r/m64, r64, imm8*	MRI	Valid	N.E.	Shift *r/m64* to right *imm8* places while shifting bits from *r64* in from the left.
0F AD /r	SHRD *r/m32, r32,* CL	MRC	Valid	Valid	Shift *r/m32* to right CL places while shifting bits from *r32* in from the left.
REX.W + 0F AD /r	SHRD *r/m64, r64,* CL	MRC	Valid	N.E.	Shift *r/m64* to right CL places while shifting bits from *r64* in from the left.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
```

```
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, COUNT – 1]; (* Last bit shifted out on exit *)
                FOR i ← 0 TO SIZE – 1 – COUNT
                    DO
                        BIT[DEST, i] ← BIT[DEST, i + COUNT];
                    OD;
                FOR i ← SIZE – COUNT TO SIZE – 1
                    DO
                        BIT[DEST,i] ← BIT[SRC, i + COUNT – SIZE];
                    OD;
        FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions

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

SHUFPD—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle two pairs of double-precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.
EVEX.128.66.0F.W1 C6 /r ib VSHUFPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Shuffle two paris of double-precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 C6 /r ib VSHUFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Shuffle four paris of double-precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F	Shuffle eight paris of double-precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	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

Selects a double-precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double-precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

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 updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

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 select controls are the bit 3:0 of the imm8 byte, imm8[7:4) are ignored.

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. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

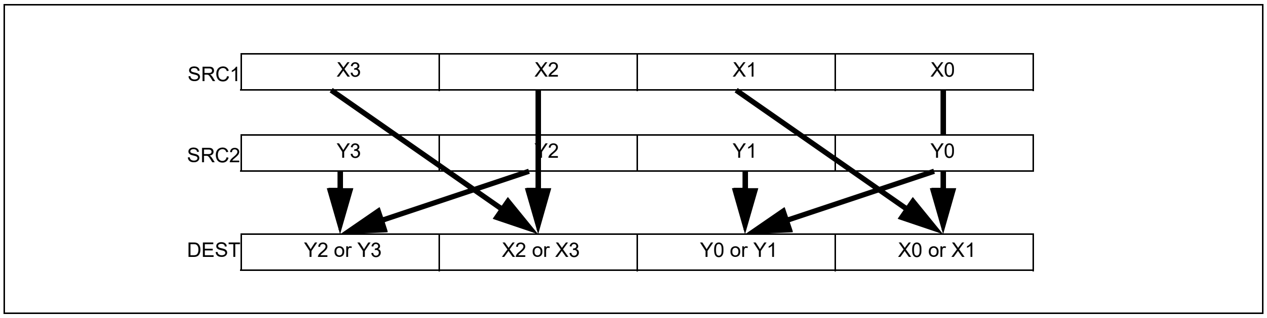

Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of DP FP Values

Operation

VSHUFPD (EVEX encoded versions when SRC2 is a vector register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] ← SRC2[63:0]
    ELSE TMP_DEST[127:64] ← SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] ← SRC2[191:128]
        ELSE TMP_DEST[255:192] ← SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] ← SRC2[319:256]
        ELSE TMP_DEST[383:320] ← SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] ← SRC2[447:384]
        ELSE TMP_DEST[511:448] ← SRC2[511:448] FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
```

```
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFPD (EVEX encoded versions when SRC2 is memory)

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

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] ← TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] ← TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] ← TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] ← TMP_SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] ← TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] ← TMP_SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] ← TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] ← TMP_SRC2[511:448] FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
```

```
                ELSE *zeroing-masking*                    ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFPD (VEX.256 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ←SRC1[63:0]
    ELSE DEST[63:0] ←SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ←SRC2[63:0]
    ELSE DEST[127:64] ←SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] ←SRC1[191:128]
    ELSE DEST[191:128] ←SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] ←SRC2[191:128]
    ELSE DEST[255:192] ←SRC2[255:192] FI;
DEST[MAXVL-1:256] (Unmodified)
```

VSHUFPD (VEX.128 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ←SRC1[63:0]
    ELSE DEST[63:0] ←SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ←SRC2[63:0]
    ELSE DEST[127:64] ←SRC2[127:64] FI;
DEST[MAXVL-1:128] ←0
```

VSHUFPD (128-bit Legacy SSE version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ←SRC1[63:0]
    ELSE DEST[63:0] ←SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ←SRC2[63:0]
    ELSE DEST[127:64] ←SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSHUFPD __m512d _mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_maskz_shuffle_pd( __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);
VSHUFPD __m256d _mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d _mm256_maskz_shuffle_pd( __mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);
VSHUFPD __m128d _mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d _mm_maskz_shuffle_pd( __mmask8 k, __m128d a, __m128d b, int imm);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	C	V/V	AVX512F	Select from quadruplet of single-precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	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

Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source 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 updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

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. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

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. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The 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. Imm8[7:0] provides 4 select controls for each element of the destination.

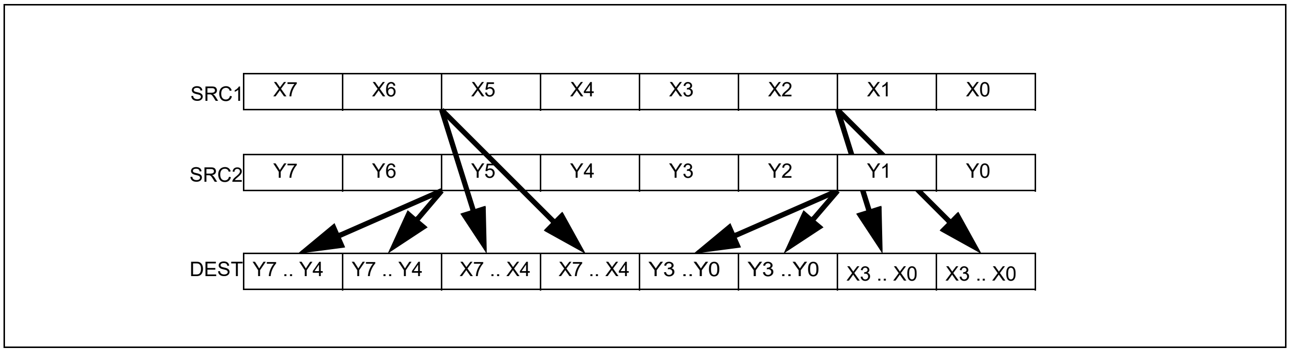

Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP ←SRC[31:0];
    1:    TMP ←SRC[63:32];
    2:    TMP ←SRC[95:64];
    3:    TMP ←SRC[127:96];
ESAC;
RETURN TMP
}
```

VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)

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

TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ←Select4(SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
```

```
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSHUFPS (EVEX encoded versions when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ←Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

DEST[MAXVL-1:VL] ← 0

VSHUFPS (VEX.256 encoded version)

DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ←Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ←Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ←Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ←Select4(SRC2[255:128], imm8[7:6]);
DEST[MAXVL-1:256] ←0

VSHUFPS (VEX.128 encoded version)

DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] ←0

SHUFPS (128-bit Legacy SSE version)

DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

SIDT—Store Interrupt Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /1	SIDT *m*	M	Valid	Valid	Store IDTR to *m.*

Instruction Operand Encoding

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

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SIDT
    THEN
        IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
            THEN
                DEST[0:15] ← IDTR(Limit);
                DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)
            ELSE (* 64-bit Mode *)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#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 CPL = 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.
	If CR4.UMIP = 1.
#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.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /0	SLDT *r/m16*	M	Valid	Valid	Stores segment selector from LDTR in *r/m16*.
REX.W + 0F 00 /0	SLDT *r64/m16*	M	Valid	Valid	Stores segment selector from LDTR in *r64/m16*.

Instruction Operand Encoding

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

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

Operation

```
DEST ← LDTR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#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 CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

SMSW—Store Machine Status Word

Opcode*	**Instruction**	**Op/ En**	**64-Bit Mode**	**Compat/ Leg Mode**	**Description**
0F 01 /4	SMSW *r/m16*	M	Valid	Valid	Store machine status word to *r/m16.*
0F 01 /4	SMSW *r32/m16*	M	Valid	Valid	Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined.
REX.W + 0F 01 /4	SMSW *r64/m16*	M	Valid	Valid	Store machine status word in low-order 16 bits of *r64/m16*; high-order 16 bits of *r32* are undefined.

Instruction Operand Encoding

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

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW r16 operand size 16, store CR0[15:0] in r16
- SMSW r32 operand size 32, zero-extend CR0[31:0], and store in r32
- SMSW r64 operand size 64, zero-extend CR0[63:0], and store in r64
- SMSW m16 operand size 16, store CR0[15:0] in m16
- SMSW m16 operand size 32, store CR0[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CR0[15:0] in m16 (not m64)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs if CR4.UMIP = 0. It is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on IA-32 and Intel 64 processors beginning with the Intel386 processors should use the MOV CR instruction to load the machine status word.

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

Operation

```
DEST ← CR0[15:0];
(* Machine status word *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#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 CPL = 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(0)	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.
	If CR4.UMIP = 1.
#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.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

SQRTPD—Square Root of Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Computes Square Roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the square roots of the two, four or eight packed double-precision floating-point values in the source operand (the second operand) stores the packed double-precision floating-point results in the destination operand (the first operand).

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 ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The 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 source operand second source operand or a 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 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.

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

Operation

VSQRTPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← SQRT(SRC[63:0])
                ELSE DEST[i+63:i] ← SQRT(SRC[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
```

VSQRTPD (VEX.256 encoded version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[191:128] ←SQRT(SRC[191:128])
DEST[255:192] ←SQRT(SRC[255:192])
DEST[MAXVL-1:256] ← 0
.
```

VSQRTPD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[MAXVL-1:128] ←0
```

SQRTPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);
VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);
VSQRTPD __m512d _mm512_maskz_sqrt_round_pd( __mmask8 k, __m512d a, int r);
VSQRTPD __m256d _mm256_sqrt_pd (__m256d a);
VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);
VSQRTPD __m256d _mm256_maskz_sqrt_pd( __mmask8 k, __m256d a, int r);
SQRTPD __m128d _mm_sqrt_pd (__m128d a);
VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);
VSQRTPD __m128d _mm_maskz_sqrt_pd( __mmask8 k, __m128d a, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

SQRTPS—Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 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 updated according to the writemask.

VEX.256 encoded version: The 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 source operand second source operand or a 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 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.

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

Operation

VSQRTPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← SQRT(SRC[31:0])
                ELSE DEST[i+31:i] ← SQRT(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
```

VSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[159:128] ←SQRT(SRC[159:128])
DEST[191:160] ←SQRT(SRC[191:160])
DEST[223:192] ←SQRT(SRC[223:192])
DEST[255:224] ←SQRT(SRC[255:224])
```

VSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[MAXVL-1:128] ←0
```

SQRTPS (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.LIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

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

Description

Computes the square root of 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 operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

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

Operation

VSQRTSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SQRT(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
```

VSQRTSD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

SQRTSD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

SQRTSS—Compute Square Root of Scalar Single-Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

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

Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

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 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

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

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

Operation

VSQRTSS (EVEX encoded version)

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

VSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

SQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss(__mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

STAC—Set AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CB STAC	ZO	V/V	SMAP	Set the AC flag in the EFLAGS register.

Instruction Operand Encoding

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

Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register.

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

Operation

```
EFLAGS.AC ← 1;
```

Flags Affected

AC set. 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 STAC 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.

STC—Set Carry Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	ZO	Valid	Valid	Set CF flag.

Instruction Operand Encoding

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

Description

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

Operation

CF ← 1;

Flags Affected

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

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

STD—Set Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FD	STD	ZO	Valid	Valid	Set DF flag.

Instruction Operand Encoding

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

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

DF ← 1;

Flags Affected

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

Exceptions (All Operating Modes)

#UD   If the LOCK prefix is used.

STI—Set Interrupt Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FB	STI	ZO	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

Instruction Operand Encoding

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

Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. This allows the processor to respond to maskable hardware interrupts.

If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI. (The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure or subroutine. For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized. No interrupts can be recognized if an execution of CLI immediately follow such an execution of STI.) The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and nonmaskable interrupts (NMIs). However, NMIs (and system-management interrupts) may be inhibited on the instruction boundary following an execution of STI that begins with IF = 0.

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, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-18 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

Table 4-18. Decision Table for STI Results

Mode	IOPL	EFLAGS.VIP	STI Result
Real-address	X[1]	X	IF = 1
Protected, not PVI[2]	≥ CPL	X	IF = 1
	< CPL	X	#GP fault
Protected, PVI[3]	3	X	IF = 1
	0–2	0	VIF = 1
		1	#GP fault
Virtual-8086, not VME[3]	3	X	IF = 1
	0–2	X	#GP fault
Virtual-8086, VME[3]	3	X	IF = 1
	0–2	0	VIF = 1
		1	#GP fault

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  (* Executing in real-address mode *)
    THEN IF ← 1; (* Set Interrupt Flag *)
    ELSE
        IF IOPL ≥ CPL  (* CPL = 3 if EFLAGS.VM = 1 *)
            THEN IF ← 1; (* Set Interrupt Flag *)
            ELSE
                IF VME mode OR PVI mode
                    THEN
                        IF EFLAGS.VIP = 0
                            THEN VIF ← 1; (* Set Virtual Interrupt Flag *)
                            ELSE #GP(0);
                        FI;
                    ELSE #GP(0);
                FI;
        FI;
FI;
```

Flags Affected

Either the IF flag or the VIF flag is set to 1. 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 EFLAGS.VIP = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL is less than 3 and VME mode is not active.
	If IOPL is less than 3 and EFLAGS.VIP = 1.
#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.

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /3 STMXCSR *m32*	M	V/V	SSE	Store contents of MXCSR register to *m32*.
VEX.LZ.0F.WIG AE /3 VSTMXCSR *m32*	M	V/V	AVX	Store contents of MXCSR register to *m32*.

Instruction Operand Encoding

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

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

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

VEX.L must be 0, otherwise instructions will #UD.

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

Operation

```
m32 ← MXCSR;
```

Intel C/C++ Compiler Intrinsic Equivalent

_mm_getcsr(void)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L= 1,
	If VEX.vvvv ≠ 1111B.

STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Opcode	**Instruction**	**Op/ En**	**64-Bit Mode**	**Compat/ Leg Mode**	**Description**
AA	STOS *m8*	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS *m16*	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS *m32*	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS *m64*	NA	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	NA	Valid	N.E.	Store RAX at address RDI or EDI.

Instruction Operand Encoding

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

Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

NOTE: To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Operation

```
Non-64-bit Mode:

IF (Byte store)
   THEN
       DEST ← AL;
           THEN IF DF = 0
               THEN (E)DI ← (E)DI + 1;
               ELSE (E)DI ← (E)DI – 1;
           FI;
   ELSE IF (Word store)
       THEN
           DEST ← AX;
               THEN IF DF = 0
                   THEN (E)DI ← (E)DI + 2;
                   ELSE (E)DI ← (E)DI – 2;
               FI;
       FI;
   ELSE IF (Doubleword store)
       THEN
           DEST ← EAX;
               THEN IF DF = 0
                   THEN (E)DI ← (E)DI + 4;
                   ELSE (E)DI ← (E)DI – 4;
               FI;
       FI;
FI;

64-bit Mode:

IF (Byte store)
   THEN
       DEST ← AL;
           THEN IF DF = 0
               THEN (R|E)DI ← (R|E)DI + 1;
               ELSE (R|E)DI ← (R|E)DI – 1;
           FI;
   ELSE IF (Word store)
       THEN
           DEST ← AX;
```

```
                THEN IF DF = 0
                    THEN (R|E)DI ← (R|E)DI + 2;
                    ELSE (R|E)DI ← (R|E)DI – 2;
                FI;
        FI;
    ELSE IF (Doubleword store)
        THEN
            DEST ← EAX;
                THEN IF DF = 0
                    THEN (R|E)DI ← (R|E)DI + 4;
                    ELSE (R|E)DI ← (R|E)DI – 4;
                FI;
        FI;
    ELSE IF (Quadword store using REX.W )
        THEN
            DEST ← RAX;
                THEN IF DF = 0
                    THEN (R|E)DI ← (R|E)DI + 8;
                    ELSE (R|E)DI ← (R|E)DI – 8;
                FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the ES segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

STR—Store Task Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /1	STR *r/m16*	M	Valid	Valid	Stores segment selector from TR in *r/m16*.

Instruction Operand Encoding

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

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

```
DEST ← TR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#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 STR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C *ib*	SUB AL, *imm8*	I	Valid	Valid	Subtract *imm8* from AL.
2D *iw*	SUB AX, *imm16*	I	Valid	Valid	Subtract *imm16* from AX.
2D *id*	SUB EAX, *imm32*	I	Valid	Valid	Subtract *imm32* from EAX.
REX.W + 2D *id*	SUB RAX, *imm32*	I	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from RAX.
80 /5 *ib*	SUB *r/m8, imm8*	MI	Valid	Valid	Subtract *imm8* from *r/m8.*
REX + 80 /5 *ib*	SUB *r/m8*, imm8*	MI	Valid	N.E.	Subtract *imm8* from *r/m8.*
81 /5 *iw*	SUB *r/m16, imm16*	MI	Valid	Valid	Subtract *imm16* from *r/m16.*
81 /5 *id*	SUB *r/m32, imm32*	MI	Valid	Valid	Subtract *imm32* from *r/m32.*
REX.W + 81 /5 *id*	SUB *r/m64, imm32*	MI	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from *r/m64.*
83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16.*
83 /5 *ib*	SUB *r/m32, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m32.*
REX.W + 83 /5 *ib*	SUB *r/m64, imm8*	MI	Valid	N.E.	Subtract sign-extended *imm8* from *r/m64.*
28 /*r*	SUB *r/m8, r8*	MR	Valid	Valid	Subtract *r8* from *r/m8.*
REX + 28 /*r*	SUB *r/m8*, r8**	MR	Valid	N.E.	Subtract *r8* from *r/m8.*
29 /*r*	SUB *r/m16, r16*	MR	Valid	Valid	Subtract *r16* from *r/m16.*
29 /*r*	SUB *r/m32, r32*	MR	Valid	Valid	Subtract *r32* from *r/m32.*
REX.W + 29 /*r*	SUB *r/m64, r64*	MR	Valid	N.E.	Subtract *r64* from *r/m64.*
2A /*r*	SUB *r8, r/m8*	RM	Valid	Valid	Subtract *r/m8* from *r8.*
REX + 2A /*r*	SUB *r8*, r/m8**	RM	Valid	N.E.	Subtract *r/m8* from *r8.*
2B /*r*	SUB *r16, r/m16*	RM	Valid	Valid	Subtract *r/m16* from *r16.*
2B /*r*	SUB *r32, r/m32*	RM	Valid	Valid	Subtract *r/m32* from *r32.*
REX.W + 2B /*r*	SUB *r64, r/m64*	RM	Valid	N.E.	Subtract *r/m64* from *r64.*

NOTES:

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

Instruction Operand Encoding

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

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

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

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

Operation

```
DEST ← (DEST – SRC);
```

Flags Affected

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

Protected Mode Exceptions

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

SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Subtract 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 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Subtract 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 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Subtract 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	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the two, four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

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

VSUBPD (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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSUBPD (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    EST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i];
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSUBPD (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
```

VSUBPD (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

SUBPD (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

VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F	Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

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

VSUBPS (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VSUBPS (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[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VSUBPS (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
```

VSUBPS (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

SUBPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256 _mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128 _mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

SUBSD—Subtract Scalar Double-Precision Floating-Point Value

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

Instruction Operand Encoding

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

Description

Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

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

VEX.128 and EVEX encoded versions: 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 encoded version: The low quadword element of the destination operand is updated according to the writemask.

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

Operation

VSUBSD (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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
```

VSUBSD (VEX.128 encoded version)

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

SUBSD (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

```
VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

SUBSS—Subtract Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.LIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

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

Description

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

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

VEX.128 and EVEX encoded versions: 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 low doubleword element of the destination operand is updated according to the writemask.

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

Operation

VSUBSS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
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
```

VSUBSS (VEX.128 encoded version)

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

SUBSS (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

```
VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F8	SWAPGS	ZO	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

Instruction Operand Encoding

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

Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

Operation

```
IF CS.L ≠ 1 (* Not in 64-Bit Mode *)
    THEN
        #UD; FI;

IF CPL ≠ 0
    THEN #GP(0); FI;

tmp ← GS.base;
GS.base ← IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE ← tmp;
```

Flags Affected

None

Protected Mode Exceptions

#UD If Mode ≠ 64-Bit.

Real-Address Mode Exceptions

#UD If Mode ≠ 64-Bit.

Virtual-8086 Mode Exceptions

#UD If Mode ≠ 64-Bit.

Compatibility Mode Exceptions

#UD	If Mode ≠ 64-Bit.

64-Bit Mode Exceptions

#GP(0)	If CPL ≠ 0.
#UD	If the LOCK prefix is used.

SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	ZO	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

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

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

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

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD;
FI;

RCX ← RIP;                              (* Will contain address of next instruction *)
RIP ← IA32_LSTAR;
R11 ← RFLAGS;
RFLAGS ← RFLAGS AND NOT(IA32_FMASK);

CS.Selector ← IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
```

```
CS.L ← 1;                            (* Entry is to 64-bit mode *)
CS.D ← 0;                            (* Required if CS.L = 1 *)
CS.G ← 1;                            (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← IA32_STAR[47:32] + 8;  (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                         (* Flat segment *)
SS.Limit ← FFFFFH;                   (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                         (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                            (* 32-bit stack segment *)
SS.G ← 1;                            (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.

SYSENTER—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	ZO	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

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

Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

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

Operation

```
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM ← 0;                          (* Ensures protected mode execution *)
RFLAGS.IF ← 0;                          (* Mask interrupts *)
IF in IA-32e mode
    THEN
        RSP ← IA32_SYSENTER_ESP;
        RIP ← IA32_SYSENTER_EIP;
ELSE
        ESP ← IA32_SYSENTER_ESP[31:0];
        EIP ← IA32_SYSENTER_EIP[31:0];
FI;

CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
                                        (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
IF in IA-32e mode
    THEN
        CS.L ← 1;                       (* Entry is to 64-bit mode *)
        CS.D ← 0;                       (* Required if CS.L = 1 *)
    ELSE
        CS.L ← 0;
        CS.D ← 1;                       (* 32-bit code segment*)
FI;
CS.G ← 1;                               (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8;          (* SS just above CS *)
```

```
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                    (* Flat segment *)
SS.Limit ← FFFFFH;              (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                    (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                       (* 32-bit stack segment*)
SS.G ← 1;                       (* 4-KByte granularity *)
```

Flags Affected

VM, IF (see Operation above)

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Instruction Operand Encoding

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

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
   THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
       THEN
           SYSENTER/SYSEXIT_Not_Supported; FI;
       ELSE
           SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

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

Operation

```
IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;

IF operand size is 64-bit
    THEN    (* Return to 64-bit mode *)
        RSP ← RCX;
        RIP ← RDX;
    ELSE    (* Return to protected mode or compatibility mode *)
        RSP ← ECX;
        RIP ← EDX;
FI;

IF operand size is 64-bit                    (* Operating system provides CS; RPL forced to 3 *)
    THEN CS.Selector ← IA32_SYSENTER_CS[15:0] + 32;
    ELSE CS.Selector ← IA32_SYSENTER_CS[15:0] + 16;
FI;
CS.Selector ← CS.Selector OR 3;              (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                                 (* Flat segment *)
CS.Limit ← FFFFFH;                           (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                                (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF operand size is 64-bit
    THEN    (* return to 64-bit mode *)
        CS.L ← 1;                            (* 64-bit code segment *)
        CS.D ← 0;                            (* Required if CS.L = 1 *)
    ELSE    (* return to protected mode or compatibility mode *)
        CS.L ← 0;
        CS.D ← 1;                            (* 32-bit code segment*)
FI;
CS.G ← 1;                                    (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← CS.Selector + 8;               (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                 (* Flat segment *)
SS.Limit ← FFFFFH;                           (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                 (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                                    (* 32-bit stack segment*)
SS.G ← 1;                                    (* 4-KByte granularity *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
	If CPL $\neq$ 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	The SYSEXIT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS = 0.
	If CPL $\neq$ 0.
	If RCX or RDX contains a non-canonical address.
#UD	If the LOCK prefix is used.

SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	ZO	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + 0F 07	SYSRET	ZO	Valid	Invalid	Return to 64-bit mode from fast system call

Instruction Operand Encoding

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

Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.[1] With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, "Interrupt Stack Table," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

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

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL ≠ 0) THEN #GP(0); FI;

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        IF (RCX is not canonical) THEN #GP(0);
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;               (* Clear RF, VM, reserved bits; set bit 1 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
FI;
CS.Selector ← CS.Selector OR 3;             (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                                (* Flat segment *)
CS.Limit ← FFFFFH;                          (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                               (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L ← 1;                           (* 64-bit code segment *)
        CS.D ← 0;                           (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L ← 0;                           (* Compatibility mode *)
        CS.D ← 1;                           (* 32-bit code segment *)
FI;
CS.G ← 1;                                   (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3;    (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                (* Flat segment *)
SS.Limit ← FFFFFH;                          (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                                   (* 32-bit stack segment*)
SS.G ← 1;                                   (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.
#GP(0)	If CPL ≠ 0.
	If the return is to 64-bit mode and RCX contains a non-canonical address.

TEST—Logical Compare

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A8 *ib*	TEST AL, i*mm8*	I	Valid	Valid	AND *imm8* with AL; set SF, ZF, PF according to result.
A9 *iw*	TEST AX, i*mm16*	I	Valid	Valid	AND *imm16* with AX; set SF, ZF, PF according to result.
A9 *id*	TEST EAX, i*mm32*	I	Valid	Valid	AND *imm32* with EAX; set SF, ZF, PF according to result.
REX.W + A9 *id*	TEST RAX, i*mm32*	I	Valid	N.E.	AND *imm32* sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 *ib*	TEST *r/m8, imm8*	MI	Valid	Valid	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
REX + F6 /0 *ib*	TEST *r/m8*, imm8*	MI	Valid	N.E.	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
F7 /0 *iw*	TEST *r/m16, imm16*	MI	Valid	Valid	AND *imm16* with *r/m16*; set SF, ZF, PF according to result.
F7 /0 *id*	TEST *r/m32, imm32*	MI	Valid	Valid	AND *imm32* with *r/m32*; set SF, ZF, PF according to result.
REX.W + F7 /0 *id*	TEST *r/m64, imm32*	MI	Valid	N.E.	AND *imm32* sign-extended to 64-bits with *r/m64*; set SF, ZF, PF according to result.
84 /*r*	TEST *r/m8, r8*	MR	Valid	Valid	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
REX + 84 /*r*	TEST *r/m8*, r8**	MR	Valid	N.E.	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
85 /*r*	TEST *r/m16, r16*	MR	Valid	Valid	AND *r16* with *r/m16*; set SF, ZF, PF according to result.
85 /*r*	TEST *r/m32, r32*	MR	Valid	Valid	AND *r32* with *r/m32*; set SF, ZF, PF according to result.
REX.W + 85 /*r*	TEST *r/m64, r64*	MR	Valid	N.E.	AND *r64* with *r/m64*; set SF, ZF, PF according to result.

NOTES:

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

Instruction Operand Encoding

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

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

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

Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);

IF TEMP = 0
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI:

PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(* AF is undefined *)
```

Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the "Operation" section above). The state of the AF flag is undefined.

Protected Mode Exceptions

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

TPAUSE—Timed PAUSE

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 TPAUSE r32, <edx>, <eax>	A	V/V	WAITPKG	Directs the processor to enter an implementation-dependent optimized state until the TSC reaches the value in EDX:EAX.

Instruction Operand Encoding[1]

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r)	NA	NA	NA

Description

TPAUSE instructs the processor to enter an implementation-dependent optimized state. There are two such optimized states to choose from: light-weight power/performance optimized state, and improved power/performance optimized state. The selection between the two is governed by the explicit input register bit[0] source operand.

TPAUSE is available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. TPAUSE may be executed at any privilege level. This instruction's operation is the same in non-64-bit modes and in 64-bit mode.

Unlike PAUSE, the TPAUSE instruction will not cause an abort when used inside a transactional region, described in the chapter Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if non-zero.

Table 4-19. TPAUSE Input Register Bit Definitions

Bit Value	State Name	Wakeup Time	Power Savings	Other Benefits
bit[0] = 0	C0.2	Slower	Larger	Improves performance of the other SMT thread(s) on the same core.
bit[0] = 1	C0.1	Faster	Smaller	NA
bits[31:1]	NA	NA	NA	Reserved

The instruction execution wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value.

Prior to executing the TPAUSE instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32-bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[1] — Reserved.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.

If the processor that executed a TPAUSE instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the read-set range within the transactional region, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal.

1. The Mod field of the ModR/M byte must have value 11B.

Other implementation-dependent events may cause the processor to exit the implementation-dependent optimized state proceeding to the instruction following TPAUSE. In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0). It should be noted that if maskable-interrupts are inhibited execution will proceed to the instruction following TPAUSE.

Operation

```
os_deadline ← TSC+(IA32_MWAIT_CONTROL[31:2]<<2)
instr_deadline ← UINT64(EDX:EAX)

IF os_deadline < instr_deadline:
    deadline ← os_deadline
    using_os_deadline ← 1
ELSE:
    deadline ← instr_deadline
    using_os_deadline ← 0

WHILE TSC < deadline:
    implementation_dependent_optimized_state(Source register, deadline, IA32_UMWAIT_CONTROL[0])

IF using_os_deadline AND TSC > deadline:
    RFLAGS.CF ← 1
ELSE:
    RFLAGS.CF ← 0

RFLAGS.AF,PF,SF,ZF,OF ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

TPAUSE uint8_t _tpause(uint32_t control, uint64_t counter);

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#GP(0)	If src[31:1] != 0.
	If CR4.TSD = 1 and CPL != 0.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

TZCNT — Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BC /r TZCNT *r16, r/m16*	A	V/V	BMI1	Count the number of trailing zero bits in *r/m16*, return result in *r16*.
F3 0F BC /r TZCNT *r32, r/m32*	A	V/V	BMI1	Count the number of trailing zero bits in *r/m32*, return result in *r32.*
F3 REX.W 0F BC /r TZCNT *r64, r/m64*	A	V/N.E.	BMI1	Count the number of trailing zero bits in *r/m64*, return result in *r64*.

Instruction Operand Encoding

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

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

```
temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )

    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 is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```
TZCNT:      unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT:      unsigned __int64 _tzcnt_u64(unsigned __int64 src);
```

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

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

UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD 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 2E /r VUCOMISD 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.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

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

Description

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

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

Note: 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

(V)UCOMISD (all versions)

```
RESULT ← UnorderedCompare(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

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2E /r UCOMISS 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.0F.WIG 2E /r VUCOMISS 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.LIG.0F.W0 2E /r VUCOMISS 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	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

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

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

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

Note: 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

(V)UCOMISS (all versions)

```
RESULT ←  UnorderedCompare(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

VUCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomineq_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

UD—Undefined Instruction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F FF /r	UD0[1] *r32, r/m32*	RM	Valid	Valid	Raise invalid opcode exception.
0F B9 /r	UD1 *r32, r/m32*	RM	Valid	Valid	Raise invalid opcode exception.
0F 0B	UD2	ZO	Valid	Valid	Raise invalid opcode exception.

NOTES:

1. Some older processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid-opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.

Instruction Operand Encoding

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

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

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

Operation

```
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	Raises an invalid opcode exception in all operating modes.

UMONITOR—User Level Set Up Monitor Address

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 UMONITOR r16/r32/r64	A	V/V	WAITPKG	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is contained in r16/r32/r64.

Instruction Operand Encoding[1]

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r)	NA	NA	NA

Description

The UMONITOR instruction arms address monitoring hardware using an address specified in the source register (the address range that the monitoring hardware checks for store operations can be determined by using the CPUID monitor leaf function, EAX=05H). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by UMWAIT.

The content of the source register is an effective address. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used. The address range must use memory of the write-back type. Only write-back memory is guaranteed to correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, "Multiple-Processor Management" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The UMONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, UMONITOR sets the A-bit but not the D-bit in page tables.

UMONITOR and UMWAIT are available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. UMONITOR and UMWAIT may be executed at any privilege level. Except for the width of the source register, the instruction's operation is the same in non-64-bit modes and in 64-bit mode.

UMONITOR does not interoperate with the legacy MWAIT instruction. If UMONITOR was executed prior to executing MWAIT and following the most recent execution of the legacy MONITOR instruction, MWAIT will not enter an optimized state. Execution will continue to the instruction following MWAIT.

The UMONITOR instruction causes a transactional abort when used inside a transactional region.

The width of the source register (16b, 32b or 64b) is determined by the effective addressing width, which is affected in the standard way by the machine mode settings and 67 prefix.

Operation

UMONITOR sets up an address range for the monitor hardware using the content of source register as an effective address and puts the monitor hardware in armed state. A store to the specified address range will trigger the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

UMONITOR void _umonitor(void *address);

Numeric Exceptions

None

1. The Mod field of the ModR/M byte must have value 11B.

Protected Mode Exceptions

#GP(0)	If the specified segment is not SS and the source register is outside the specified segment limit.
	If the specified segment register contains a NULL segment selector.
#SS(0)	If the specified segment is SS and the source register is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

Real Address Mode Exceptions

#GP	If the specified segment is not SS and the source register is outside of the effective address space from 0 to FFFFH.
#SS	If the specified segment is SS and the source register is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

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

#GP(0)	If the specified segment is not SS and the linear address is in non-canonical form.
#SS(0)	If the specified segment is SS and the source register is in non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

UMWAIT—User Level Monitor Wait

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F AE /6 UMWAIT r32, <edx>, <eax>	A	V/V	WAITPKG	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding[1]

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r)	NA	NA	NA

Description

UMWAIT instructs the processor to enter an implementation-dependent optimized state while monitoring a range of addresses. The optimized state may be either a light-weight power/performance optimized state or an improved power/performance optimized state. The selection between the two states is governed by the explicit input register bit[0] source operand.

UMWAIT is available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. UMWAIT may be executed at any privilege level. This instruction's operation is the same in non-64-bit modes and in 64-bit mode.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if nonzero.

Table 4-20. UMWAIT Input Register Bit Definitions

Bit Value	State Name	Wakeup Time	Power Savings	Other Benefits
bit[0] = 0	C0.2	Slower	Larger	Improves performance of the other SMT thread(s) on the same core.
bit[0] = 1	C0.1	Faster	Smaller	NA
bits[31:1]	NA	NA	NA	Reserved

The instruction wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value (if the monitoring hardware did not trigger beforehand).

Prior to executing the UMWAIT instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[1] — Reserved.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.

If the processor that executed a UMWAIT instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The UMWAIT instruction causes a transactional abort when used inside a transactional region.

The UMWAIT instruction operates with the UMONITOR instruction. The two instructions allow the definition of an address at which to wait (UMONITOR) and an implementation-dependent optimized operation to perform while waiting (UMWAIT). The execution of UMWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by UMONITOR. The UMWAIT instruction will not wait (will not enter an implementation-dependent optimized state) if any of the

1. The Mod field of the ModR/M byte must have value 11B.

following instructions were executed before UMWAIT and after the most recent execution of UMONITOR: IRET, MONITOR, SYSEXIT, SYSRET, and far RET (the last if it is changing CPL).

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the address range armed by the UMONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0).

Following exit from the implementation-dependent-optimized state, control passes to the instruction after the UMWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction.

Unlike the HLT instruction, the UMWAIT instruction does not restart at the UMWAIT instruction following the handling of an SMI.

If the preceding UMONITOR instruction did not successfully arm an address range or if UMONITOR was not executed prior to executing UMWAIT and following the most recent execution of the legacy MONITOR instruction (UMWAIT does not interoperate with MONITOR), then the processor will not enter an optimized state. Execution will continue to the instruction following UMWAIT.

A store to the address range armed by the UMONITOR instruction will cause the processor to exit UMWAIT if either the store was originated by other processor agents or the store was originated by a non-processor agent.

Operation

```
os_deadline ← TSC+(IA32_MWAIT_CONTROL[31:2]<<2)
instr_deadline ← UINT64(EDX:EAX)

IF os_deadline < instr_deadline:
    deadline ← os_deadline
    using_os_deadline ← 1
ELSE:
    deadline ← instr_deadline
    using_os_deadline ← 0

WHILE monitor hardware armed AND TSC < deadline:
    implementation_dependent_optimized_state(Source register, deadline, IA32_UMWAIT_CONTROL[0] )

IF using_os_deadline AND TSC > deadline:
    RFLAGS.CF ← 1
ELSE:
    RFLAGS.CF ← 0

RFLAGS.AF,PF,SF,ZF,OF ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
UMWAIT uint8_t _umwait(uint32_t control, uint64_t counter);
```

Numeric Exceptions

None

Exceptions (All Operating Modes)

#GP(0)	If src[31:1] != 0.
	If CR4.TSD = 1 and CPL != 0.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 15 /r VUNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 15 /r VUNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from high quadwords of 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	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

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. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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.

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.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPD (EVEX encoded versions when SRC2 is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[127:64]
    TMP_DEST[127:64] ← SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[255:192]
    TMP_DEST[255:192] ← SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[383:320]
    TMP_DEST[383:320] ← SRC2[383:320]
    TMP_DEST[447:384] ← SRC1[511:448]
    TMP_DEST[511:448] ← SRC2[511:448]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPD (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[127:64]
    TMP_DEST[127:64] ← TMP_SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[255:192]
    TMP_DEST[255:192] ← TMP_SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[383:320]
    TMP_DEST[383:320] ← TMP_SRC2[383:320]
    TMP_DEST[447:384] ← SRC1[511:448]
    TMP_DEST[511:448] ← TMP_SRC2[511:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
DEST[191:128]←SRC1[255:192]
DEST[255:192]←SRC2[255:192]
DEST[MAXVL-1:256] ←0
```

VUNPCKHPD (VEX.128 encoded version)

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

UNPCKHPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPD __m512d _mm512_unpackhi_pd(__m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

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. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

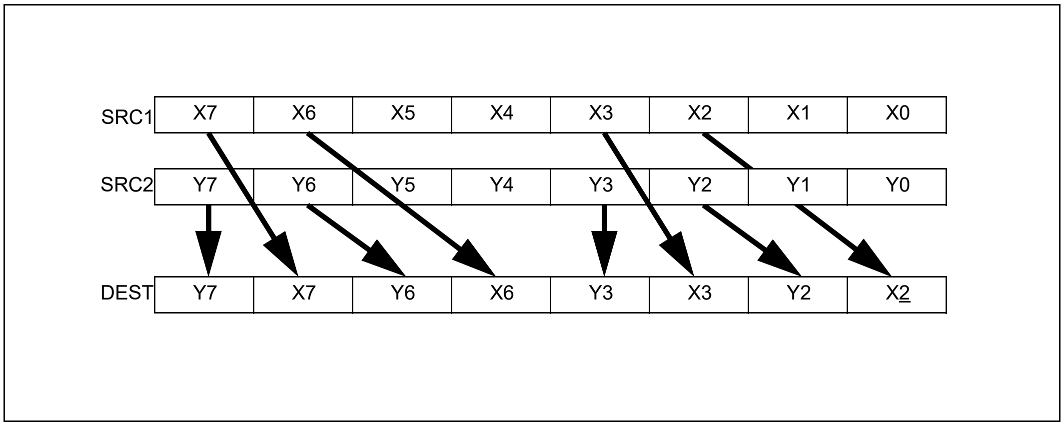

Figure 4-27. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second 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 using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPS (EVEX encoded version when SRC2 is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[95:64]
    TMP_DEST[63:32] ← SRC2[95:64]
    TMP_DEST[95:64] ← SRC1[127:96]
    TMP_DEST[127:96] ← SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[223:192]
    TMP_DEST[191:160] ← SRC2[223:192]
    TMP_DEST[223:192] ← SRC1[255:224]
    TMP_DEST[255:224] ← SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[351:320]
    TMP_DEST[319:288] ← SRC2[351:320]
    TMP_DEST[351:320] ← SRC1[383:352]
    TMP_DEST[383:352] ← SRC2[383:352]
    TMP_DEST[415:384] ← SRC1[479:448]
    TMP_DEST[447:416] ← SRC2[479:448]
    TMP_DEST[479:448] ← SRC1[511:480]
    TMP_DEST[511:480] ← SRC2[511:480]
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[95:64]
    TMP_DEST[63:32] ← TMP_SRC2[95:64]
    TMP_DEST[95:64] ← SRC1[127:96]
    TMP_DEST[127:96] ← TMP_SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[223:192]
    TMP_DEST[191:160] ← TMP_SRC2[223:192]
    TMP_DEST[223:192] ← SRC1[255:224]
    TMP_DEST[255:224] ← TMP_SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[351:320]
    TMP_DEST[319:288] ← TMP_SRC2[351:320]
    TMP_DEST[351:320] ← SRC1[383:352]
    TMP_DEST[383:352] ← TMP_SRC2[383:352]
    TMP_DEST[415:384] ← SRC1[479:448]
    TMP_DEST[447:416] ← TMP_SRC2[479:448]
    TMP_DEST[479:448] ← SRC1[511:480]
    TMP_DEST[511:480] ← TMP_SRC2[511:480]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
```

```
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[159:128] ←SRC1[223:192]
DEST[191:160] ←SRC2[223:192]
DEST[223:192] ←SRC1[255:224]
DEST[255:224] ←SRC2[255:224]
DEST[MAXVL-1:256] ← 0
```

VUNPCKHPS (VEX.128 encoded version)

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

UNPCKHPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand.

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. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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.

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.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPD (EVEX encoded versions when SRC2 is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[383:320] ← SRC2[319:256]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[511:448] ← SRC2[447:384]
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKLPD (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
         THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
         ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← TMP_SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← TMP_SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[383:320] ← TMP_SRC2[319:256]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[511:448] ← TMP_SRC2[447:384]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
         THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
         ELSE
              IF *merging-masking*                    ; merging-masking
                   THEN *DEST[i+63:i] remains unchanged*
                   ELSE *zeroing-masking*             ; zeroing-masking
                        DEST[i+63:i] ← 0
              FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKLPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
DEST[191:128] ←SRC1[191:128]
DEST[255:192] ←SRC2[191:128]
DEST[MAXVL-1:256] ← 0
```

VUNPCKLPD (VEX.128 encoded version)

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

UNPCKLPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPD __m512d _mm512_unpacklo_pd(__m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)
VUNPCKLPD __m256d _mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
VUNPCKLPD __m128d _mm_mask_unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.128.0F.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.256.0F.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

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. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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.

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.

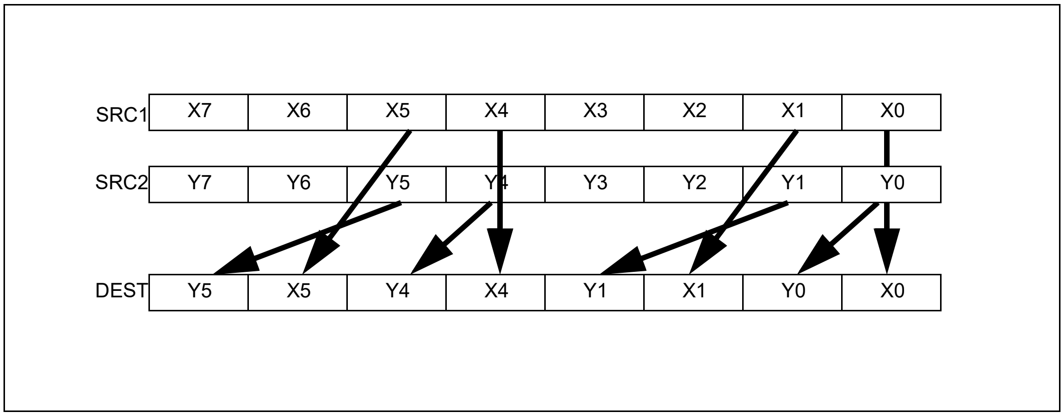

Figure 4-28. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second 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 using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[31:0]
    TMP_DEST[63:32] ← SRC2[31:0]
    TMP_DEST[95:64] ← SRC1[63:32]
    TMP_DEST[127:96] ← SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKLPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
TMP_DEST[31:0] ← SRC1[31:0]
TMP_DEST[63:32] ← TMP_SRC2[31:0]
TMP_DEST[95:64] ← SRC1[63:32]
TMP_DEST[127:96] ← TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← TMP_SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← TMP_SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← TMP_SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← TMP_SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← TMP_SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
```

```
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

UNPCKLPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[159:128] ←SRC1[159:128]
DEST[191:160] ←SRC2[159:128]
DEST[223:192] ←SRC1[191:160]
DEST[255:224] ←SRC2[191:160]
DEST[MAXVL-1:256] ← 0
```

VUNPCKLPS (VEX.128 encoded version)

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

UNPCKLPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.