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

Volume 2B: Instruction Set Reference, N-Z

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

Order Number: 253667-054US
April 2015

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting http://www.intel.com/design/literature.htm.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

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

Copyright © 1997-2015, Intel Corporation. All Rights Reserved.

CHAPTER 4
INSTRUCTION SET REFERENCE, N-Z

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, PCMP-ISTRI, 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

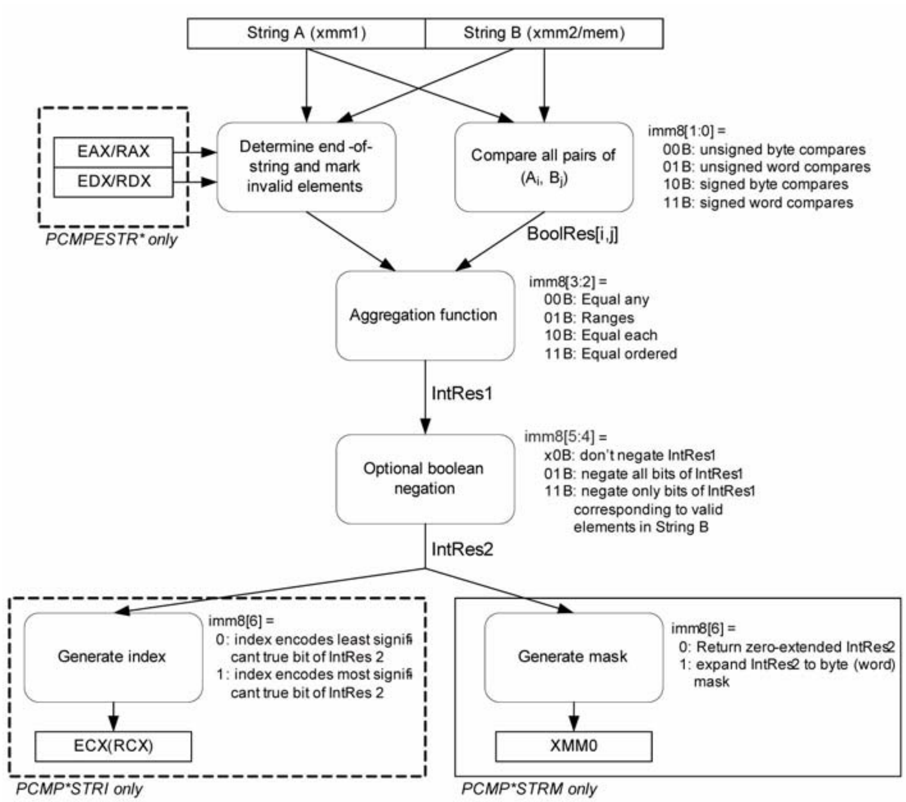

Figure 4-1. Operation of PCMPSTRx and PCMPESTRx

4.2 INSTRUCTIONS (N-Z)

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

NEG—Two's Complement Negation

Opcode	Instruction	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
90	NOP	NP	Valid	Valid	One byte no-operation instruction.
0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
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
NP	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 4-9. 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, i*mm8*	I	Valid	Valid	AL OR *imm8.*
0D *iw*	OR AX, i*mm16*	I	Valid	Valid	AX OR *imm16.*
0D *id*	OR EAX, i*mm32*	I	Valid	Valid	EAX OR *imm32.*
REX.W + 0D *id*	OR RAX, i*mm32*	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 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*	RM	V/V	SSE2	Bitwise OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG 56 /r VORPD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in *xmm2* and *xmm3/mem*.
VEX.NDS.256.66.0F.WIG 56 /r VORPD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in *ymm2* and *ymm3/mem*.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

Operation

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

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[VLMAX-1:128] ← 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]
```

Intel® C/C++ Compiler Intrinsic Equivalent

ORPD: __m128d _mm_or_pd(__m128d a, __m128d b);

VORPD: __m256d _mm256_or_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 56 /*r* ORPS *xmm1*, *xmm2/m128*	RM	V/V	SSE	Bitwise OR of *xmm1* and *xmm2/m128.*
VEX.NDS.128.0F.WIG 56 /r VORPS *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in *xmm2* and *xmm3/mem*.
VEX.NDS.256.0F.WIG 56 /r VORPS *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in *ymm2* and *ymm3/mem*.

Instruction Operand Encoding

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

Description

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

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

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

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

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

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

Operation

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

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[VLMAX-1:128] ← 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].

Intel C/C++ Compiler Intrinsic Equivalent

ORPS: __m128 _mm_or_ps (__m128 a, __m128 b);

VORPS: __m256 _mm256_or_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

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	NP	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	NP	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	NP	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
NP	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 16, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

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*	NP	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*	NP	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*	NP	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6E	OUTSB	NP	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSW	NP	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSD	NP	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
NP	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 16, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, 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 — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 1C /r[1] PABSB *mm1, mm2/m64*	RM	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*	RM	V/V	SSSE3	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
0F 38 1D /r[1] PABSW *mm1, mm2/m64*	RM	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*	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
0F 38 1E /r[1] PABSD *mm1, mm2/m64*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	V/V	AVX2	Compute the absolute value of 32-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

(V)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). (V)PABSB operates on signed bytes, (V)PABSW operates on 16-bit words, and (V)PABSD operates on signed 32-bit integers. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register, a YMM register, a 128-bit memory location, or a 256-bit memory location. The destination operand can be an MMX, an XMM or a YMM register. Both operands can be MMX registers or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16byte boundary or a general-protection exception (#GP) will be generated.

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

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

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

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

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

Operation

PABSB (with 64 bit operands)

```
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 7th bytes
Unsigned DEST[63:56] ← ABS(SRC[63:56])
```

PABSB (with 128 bit operands)

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

PABSW (with 64 bit operands)

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

PABSW (with 128 bit operands)

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

PABSD (with 64 bit operands)

```
Unsigned DEST[31:0] ← ABS(SRC[31:0])
Unsigned DEST[63:32] ← ABS(SRC[63:32])
```

PABSD (with 128 bit operands)

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

PABSB (128-bit Legacy SSE version)

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

VPABSB (VEX.128 encoded version)

DEST[127:0] ← BYTE_ABS(SRC)
DEST[VLMAX-1:128] ← 0

VPABSB (VEX.256 encoded version)

Unsigned DEST[7:0]← ABS(SRC[7:.0])
Repeat operation for 2nd through 31st bytes
Unsigned DEST[255:248] ← ABS(SRC[255:248])

PABSW (128-bit Legacy SSE version)

DEST[127:0] ← WORD_ABS(SRC)
DEST[VLMAX-1:128] (Unmodified)

VPABSW (VEX.128 encoded version)

DEST[127:0] ← WORD_ABS(SRC)
DEST[VLMAX-1:128] ← 0

VPABSW (VEX.256 encoded version)

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])

PABSD (128-bit Legacy SSE version)

DEST[127:0] ← DWORD_ABS(SRC)
DEST[VLMAX-1:128] (Unmodified)

VPABSD (VEX.128 encoded version)

DEST[127:0] ← DWORD_ABS(SRC)
DEST[VLMAX-1:128] ← 0

VPABSD (VEX.256 encoded version)

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])

Intel C/C++ Compiler Intrinsic Equivalents

PABSB: __m64 _mm_abs_pi8 (__m64 a)

(V)PABSB: __m128i _mm_abs_epi8 (__m128i a)

VPABSB: __m256i _mm256_abs_epi8 (__m256i a)

PABSW: __m64 _mm_abs_pi16 (__m64 a)

(V)PABSW: __m128i _mm_abs_epi16 (__m128i a)

VPABSW: __m256i _mm256_abs_epi16 (__m256i a)

PABSD: __m64 _mm_abs_pi32 (__m64 a)

(V)PABSD: __m128i _mm_abs_epi32 (__m128i a)

VPABSD: __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 63 /r[1] PACKSSWB *mm1, mm2/m64*	RM	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*	RM	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.
0F 6B /r[1] PACKSSDW *mm1, mm2/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG 63 /r VPACKSSWB *xmm1,xmm2, xmm3/m128*	RVM	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.NDS.128.66.0F.WIG 6B /r VPACKSSDW *xmm1,xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG 63 /r VPACKSSWB *ymm1, ymm2, ymm3/m256*	RVM	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.NDS.256.66.0F.WIG 6B /r VPACKSSDW *ymm1, ymm2, ymm3/m256*	RVM	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.

NOTES:

1. See note in Section 2.4, "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

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 4-2 for an example of the packing operation.

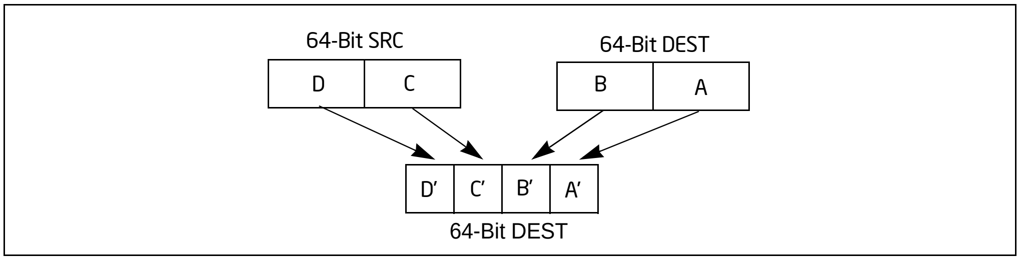

Figure 4-2. Operation of the PACKSSDW Instruction Using 64-bit Operands

The (V)PACKSSWB instruction converts 4, 8 or 16 signed word integers from the destination operand (first operand) and 4, 8 or 16 signed word integers from the source operand (second operand) into 8, 16 or 32 signed byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

The (V)PACKSSDW instruction packs 2, 4 or 8 signed doublewords from the destination operand (first operand) and 2, 4 or 8 signed doublewords from the source operand (second operand) into 4, 8 or 16 signed words in the destination operand (see Figure 4-2). If a signed doubleword integer value is beyond the range of a signed word (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed word integer value of 7FFFH or 8000H, respectively, is stored into the destination.

The (V)PACKSSWB and (V)PACKSSDW instructions operate on either 64-bit, 128-bit operands or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. 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 is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PACKSSWB (with 64-bit operands)

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

PACKSSDW (with 64-bit operands)

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

PACKSSWB instruction (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]);
```

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]);
```

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[VLMAX-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[VLMAX-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]);
```

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]);

Intel C/C++ Compiler Intrinsic Equivalents

PACKSSWB: __m64 _mm_packs_pi16(__m64 m1, __m64 m2)

(V)PACKSSWB: __m128i _mm_packs_epi16(__m128i m1, __m128i m2)

VPACKSSWB: __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)

PACKSSDW: __m64 _mm_packs_pi32 (__m64 m1, __m64 m2)

(V)PACKSSDW: __m128i _mm_packs_epi32(__m128i m1, __m128i m2)

VPACKSSDW: __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

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*	RM	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.NDS.128.66.0F38.WIG 2B /r VPACKUSDW *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F38.WIG 2B /r VPACKUSDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Convert 8 packed signed doubleword integers from *ymm2* and 8 packed signed doubleword integers from *ymm3/m128* into 16 packed unsigned word integers in *ymm1* using unsigned saturation.

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

Converts packed signed doubleword integers 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.

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

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

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[63:48] ← (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] ;
```

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[63:48] ← (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[VLMAX-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[63:48] ← (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[128:112] ← (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] ;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)PACKUSDW: __m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW:   __m256i _mm256_packus_epi32(__m256i m1, __m256i m2);
```

Flags Affected

None.

SIMD Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 67 /*r*[1] PACKUSWB *mm, mm/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG 67 /r VPACKUSWB *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG 67 /r VPACKUSWB *ymm1, ymm2, ymm3/m256*	RVM	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.

NOTES:

1. See note in Section 2.4, "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

Converts 4, 8 or 16 signed word integers from the destination operand (first operand) and 4, 8 or 16 signed word integers from the source operand (second operand) into 8, 16 or 32 unsigned byte integers and stores the result in the destination operand. (See Figure 4-2 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

The PACKUSWB instruction operates on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. 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 is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

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[VLMAX-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]);
```

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PADDB/PADDW/PADDD—Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FC /*r*[1] PADDB *mm, mm/m64*	RM	V/V	MMX	Add packed byte integers from *mm/m64* and *mm*.
66 0F FC /*r* PADDB *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
0F FD /*r*[1] PADDW *mm, mm/m64*	RM	V/V	MMX	Add packed word integers from *mm/m64* and *mm*.
66 0F FD /*r* PADDW *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
0F FE /*r*[1] PADDD *mm, mm/m64*	RM	V/V	MMX	Add packed doubleword integers from *mm/m64* and *mm*.
66 0F FE /*r* PADDD *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed byte integers from *xmm3/m128* and *xmm2*.
VEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed word integers from *xmm3/m128* and *xmm2*.
VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed doubleword integers from *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1*.

NOTES:

1. See note in Section 2.4, "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

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.

Adds the packed byte, word, doubleword, or quadword integers in the first source operand to the second source operand and stores the result in the destination operand. When a result is too large to be represented in the

8/16/32 integer (overflow), the result is wrapped around and the low bits are written to the destination element (that is, the carry is ignored).

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.

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. 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 is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

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];
```

PADDB (with 128-bit operands)

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

VPADDB (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[VLMAX-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];
```

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];
```

PADDW (with 128-bit operands)
```
DEST[15:0]  ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] ← DEST[127:112] + SRC[127:112];
```

VPADDW (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[VLMAX-1:128] ← 0
```

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];
```

PADDD (with 64-bit operands)
```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[63:32] ← DEST[63:32] + SRC[63:32];
```

PADDD (with 128-bit operands)
```
DEST[31:0] ← DEST[31:0]  + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← DEST[127:96] + SRC[127:96];
```

VPADDD (VEX.128 encoded version)
```
DEST[31:0] ← SRC1[31:0]+SRC2[31:0]
DEST[63:32] ← SRC1[63:32]+SRC2[63:32]
DEST[95:64] ← SRC1[95:64]+SRC2[95:64]
DEST[127:96] ← SRC1[127:96]+SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

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];
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDB:	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
(V)PADDB:	__m128i _mm_add_epi8 (__m128ia,__m128ib)
VPADDB:	__m256i _mm256_add_epi8 (__m256ia,__m256i b)
PADDW:	__m64 _mm_add_pi16(__m64 m1, __m64 m2)
(V)PADDW:	__m128i _mm_add_epi16 (__m128i a, __m128i b)

VPADDW: __m256i _mm256_add_epi16 (__m256i a, __m256i b)

PADDD: __m64 _mm_add_pi32(__m64 m1, __m64 m2)

(V)PADDD: __m128i _mm_add_epi32 (__m128i a, __m128i b)

VPADDD: __m256i _mm256_add_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDQ—Add Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D4 /*r*[1] PADDQ *mm1, mm2/m64*	RM	V/V	SSE2	Add quadword integer *mm2/m64* to *mm1.*
66 0F D4 /*r* PADDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Add packed quadword integers *xmm2/m128* to *xmm1.*
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed quadword integers *xmm3/m128* and *xmm2.*
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1.*

NOTES:

1. See note in Section 2.4, “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

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

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

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

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

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

PADDQ (with 128-Bit operands)

DEST[63:0] ← DEST[63:0] + SRC[63:0];
DEST[127:64] ← DEST[127:64] + SRC[127:64];

VPADDQ (VEX.128 encoded instruction)

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

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];

Intel C/C++ Compiler Intrinsic Equivalents

PADDQ: __m64 _mm_add_si64 (__m64 a, __m64 b)

(V)PADDQ: __m128i _mm_add_epi64 (__m128i a, __m128i b)

VPADDQ: __m256i _mm256_add_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EC /*r*[1] PADDSB *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F ED /*r*[1] PADDSW *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed signed byte integers from *xmm3/m128* and *xmm2* saturate the results.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed signed word integers from *xmm3/m128* and *xmm2* and saturate the results.
VEX.NDS.256.66.0F.WIG EC /r VPADDSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.NDS.256.66.0F.WIG ED /r VPADDSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.

NOTES:

1. See note in Section 2.4, “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

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.

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

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

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. 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 is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

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[VLMAX-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]);
```

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[VLMAX-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])
```

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)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DC /*r*[1] PADDUSB *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F DD /*r*[1] PADDUSW *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.
VEX.NDS.128.660F.WIG DC /r VPADDUSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed unsigned byte integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add packed unsigned word integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed unsigned byte integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add packed unsigned word integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

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.

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

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

These instructions can operate on either 64-bit, 128-bit or 256-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. 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 is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

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[VLMAX-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[VLMAX-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])
```

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)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 3A 0F /r ib[1] PALIGNR *mm1, mm2/m64, imm8*	RMI	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*	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *xmm1.*
VEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR *xmm1, xmm2, xmm3/m128, imm8*	RVMI	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.NDS.256.66.0F3A.WIG 0F /r ib VPALIGNR *ymm1, ymm2, ymm3/m256, imm8*	RVMI	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*.

NOTES:

1. See note in Section 2.4, "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
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)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, use the REX prefix to access additional registers.

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

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

VEX.256 encoded version: The first source operand is a YMM register 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.

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.

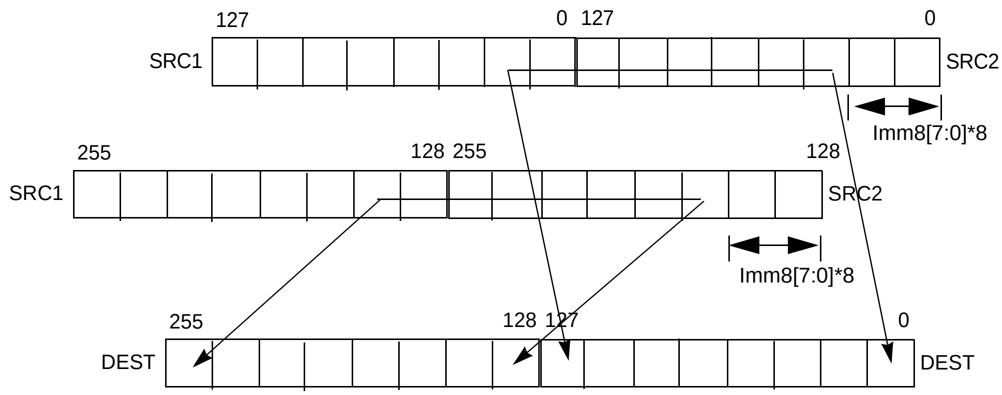

Figure 4-3. 256-bit VPALIGN Instruction Operation

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[VLMAX-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[VLMAX-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[255:128] ← temp1[127:0]
```

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)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DB /*r*[1] PAND *mm, mm/m64*	RM	V/V	MMX	Bitwise AND *mm/m64* and *mm*.
66 0F DB /*r* PAND *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Bitwise AND of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DB /r VPAND *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise AND of *xmm3/m128* and *xmm*.
VEX.NDS.256.66.0F.WIG DB /r VPAND *ymm1, ymm2, ymm3/.m256*	RVM	V/V	AVX2	Bitwise AND of *ymm2*, and *ymm3/m256* and store result in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

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, 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 a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PAND (128-bit Legacy SSE version)

```
DEST ← DEST AND SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPAND (VEX.128 encoded version)

```
DEST ← SRC1 AND SRC2
DEST[VLMAX-1:128] ← 0
```

VPAND (VEX.256 encoded instruction)

DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DF /*r*[1] PANDN *mm, mm/m64*	RM	V/V	MMX	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /*r* PANDN *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Bitwise AND NOT of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DF /r VPANDN *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise AND NOT of *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG DF /r VPANDN *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Bitwise AND NOT of *ymm2*, and *ymm3/m256* and store result in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

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, 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 a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PANDN(128-bit Legacy SSE version)

DEST ← NOT(DEST) AND SRC
DEST[VLMAX-1:128] (Unmodified)

VPANDN (VEX.128 encoded version)

DEST ← NOT(SRC1) AND SRC2
DEST[VLMAX-1:128] ← 0

VPANDN (VEX.256 encoded instruction)

DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PAUSE—Spin Loop Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 90	PAUSE	NP	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
NP	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
0F E0 /*r*[1] PAVGB *mm1, mm2/m64*	RM	V/V	SSE	Average packed unsigned byte integers from *mm2/m64* and *mm1* with rounding.
66 0F E0, /*r* PAVGB *xmm1, xmm2/m128*	RM	V/V	SSE2	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
0F E3 /*r*[1] PAVGW *mm1, mm2/m64*	RM	V/V	SSE	Average packed unsigned word integers from mm2/m64 and *mm1* with rounding.
66 0F E3 /*r* PAVGW *xmm1, xmm2/m128*	RM	V/V	SSE2	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.
VEX.NDS.128.66.0F.WIG E0 /r VPAVGB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Average packed unsigned byte integers from *xmm3/m128* and *xmm2* with rounding.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Average packed unsigned word integers from *xmm3/m128* and *xmm2* with rounding.
VEX.NDS.256.66.0F.WIG E0 /r VPAVGB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to *ymm1*.
VEX.NDS.256.66.0F.WIG E3 /r VPAVGW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Average packed unsigned word integers from *ymm2, ymm3/m256* with rounding to *ymm1*.

NOTES:

1. See note in Section 2.4, "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

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, 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 a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

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[VLMAX-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[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

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.NDS.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.NDS.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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute 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 (VLMAX-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[VLMAX-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[VLMAX-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.L = 1.
	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.NDS.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.NDS.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 (VLMAX-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 (VLMAX-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[VLMAX-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.

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.NDS.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 4-10, other bits of the immediate byte are ignored.

Table 4-10. 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 (VLMAX-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 4-11. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ *xmm1, xmm2*	**0000_0000B**
PCLMULHQLQDQ *xmm1, xmm2*	**0000_0001B**
PCLMULLQHDQ *xmm1, xmm2*	**0001_0000B**
PCLMULHQHDQ *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[VLMAX-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[VLMAX-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.

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 74 /r[1] PCMPEQB *mm, mm/m64*	RM	V/V	MMX	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /r PCMPEQB *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare packed bytes in *xmm2/m128* and *xmm1* for equality.
0F 75 /r[1] PCMPEQW *mm, mm/m64*	RM	V/V	MMX	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /r PCMPEQW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare packed words in *xmm2/m128* and *xmm1* for equality.
0F 76 /r[1] PCMPEQD *mm, mm/m64*	RM	V/V	MMX	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /r PCMPEQD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare packed doublewords in *xmm2/m128* and *xmm1* for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed bytes in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed words in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed doublewords in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	RVM	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW *ymm1, ymm2, ymm3 /m256*	RVM	V/V	AVX2	Compare packed words in *ymm3/m256* and *ymm2* for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD *ymm1, ymm2, ymm3 /m256*	RVM	V/V	AVX2	Compare packed doublewords in *ymm3/m256* and *ymm2* for equality.

NOTES:

1. See note in Section 2.4, “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

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, 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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;
```

PCMPEQB (with 128-bit operands)

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] = SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-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])
```

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)

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
```

```
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[127:112] = SRC[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

VPCMPEQW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-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])
```

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)

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] = SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

VPCMPEQD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-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])
```

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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*	RM	V/V	SSE4_1	Compare packed qwords in *xmm2/m128* and *xmm1* for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed quadwords in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ *ymm1, ymm2, ymm3 /m256*	RVM	V/V	AVX2	Compare packed quadwords in *ymm3/m256* and *ymm2* for equality.

Instruction Operand Encoding

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

Description

Performs an SIMD compare 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
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;
```

VPCMPEQQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[VLMAX-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])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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.WIG 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/RCX).

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	RCX

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.WIG 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 (VLMAX-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
0F 64 /*r*[1] PCMPGTB *mm, mm/m64*	RM	V/V	MMX	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /*r* PCMPGTB *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
0F 65 /*r*[1] PCMPGTW *mm, mm/m64*	RM	V/V	MMX	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /*r* PCMPGTW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
0F 66 /*r*[1] PCMPGTD *mm, mm/m64*	RM	V/V	MMX	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /*r* PCMPGTD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed word integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed doubleword integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed word integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed doubleword integers in *ymm2* and *ymm3/m256* for greater than.

NOTES:

1. See note in Section 2.4, “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

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, 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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;
```

PCMPGTB (with 128-bit operands)

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] > SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[VLMAX-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])
```

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)

```
IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[63:48] > SRC[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

VPCMPGTW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-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])
```

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)

```
IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] > SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

VPCMPGTD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-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])
```

Intel C/C++ Compiler Intrinsic Equivalents

PCMPGTB:	__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW:	__m64 _mm_pcmpgt_pi16 (__m64 m1, __m64 m2)
DCMPGTD:	__m64 _mm_pcmpgt_pi32 (__m64 m1, __m64 m2)
(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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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*	RM	V/V	SSE4_2	Compare packed signed qwords in *xmm2/m128* and *xmm1* for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed qwords in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed qwords in *ymm2* and *ymm3/m256* for greater than.

Instruction Operand Encoding

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

Description

Performs an SIMD 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
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
```

VPCMPGTQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-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])
```

Intel C/C++ Compiler Intrinsic Equivalent

(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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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
64 bit + REX.W	xmm	xmm/m128	RCX

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 (VLMAX-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
64 bit + REX.W	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.NDS.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.NDS.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.

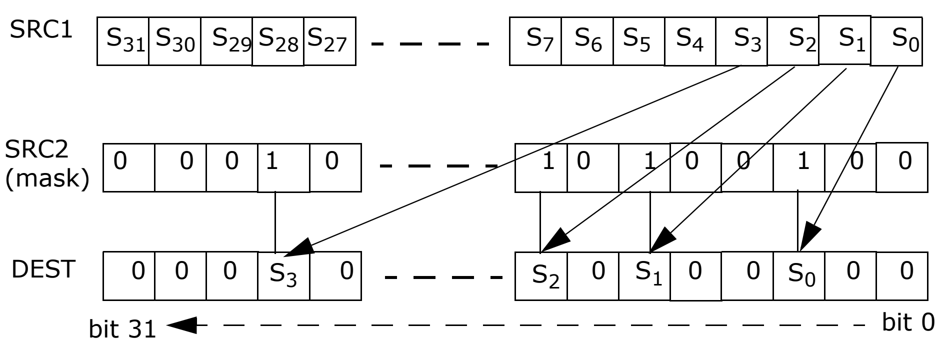

Figure 4-4. PDEP Example

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 Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.

PEXT — Parallel Bits Extract

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

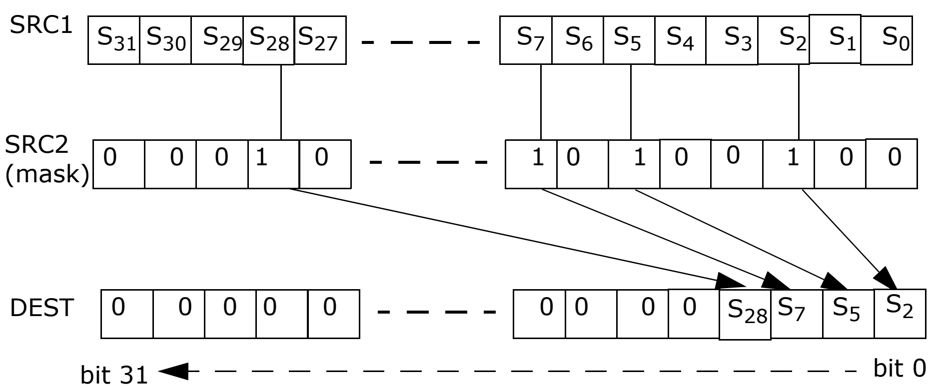

Figure 4-5. PEXT Example

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 Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD If VEX.W = 1.

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*	MRI	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*	MRI	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*	MRI	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*	MRI	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*	MRI	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*	MRI	V/i	AVX	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).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	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. 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. Attempt to execute VPEXTRQ in non-64-bit mode will cause #UD.

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:
```

(V)PEXTRTD/(V)PEXTRQ

```
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
```

(V)PEXTRB (dest=m8)

```
SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)
```

(V)PEXTRB (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

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.
	If VPEXTRQ in non-64-bit mode, VEX.W=1.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C5 /*r* ib[1] PEXTRW *reg, mm, imm8*	RMI	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*	RMI	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*	MRI	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*	RMI	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*	MRI	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.

NOTES:

1. See note in Section 2.4, “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	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
MRI	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. 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;
```

(V)PEXTRW (dest=m16)

```
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)
```

(V)PEXTRW (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

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PHADDW/PHADDD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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*.
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.NDS.128.66.0F38.WIG 01 /r VPHADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 02 /r VPHADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.NDS.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.NDS.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, "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 (VLMAX-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 (VLMAX-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.

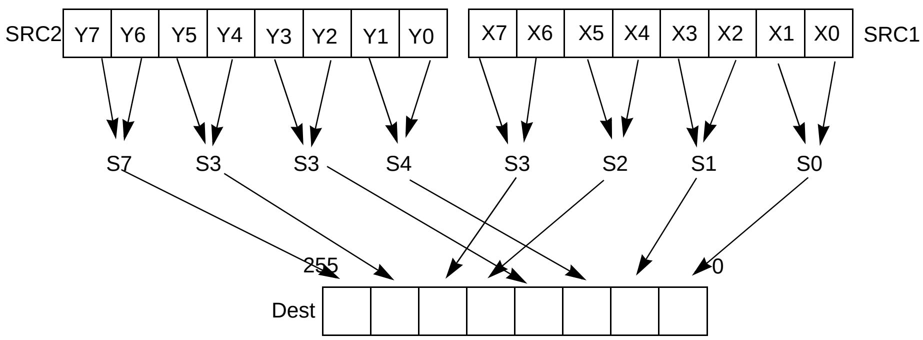

Figure 4-6. 256-bit VPHADDD Instruction Operation

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[VLMAX-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[VLMAX-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
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.NDS.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.NDS.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, "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 (VLMAX-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 (VLMAX-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[VLMAX-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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.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[VLMAX-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
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*.
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.NDS.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.NDS.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.NDS.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.NDS.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, "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 (VLMAX-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 (VLMAX-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[VLMAX-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[VLMAX-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
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.NDS.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.NDS.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, "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 (VLMAX-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 (VLMAX-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[VLMAX-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*	RMI	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*	RMI	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*	RMI	V/N. E.	SSE4_1	Insert a qword integer value from *r/m64* into the *xmm1* at the destination element specified by *imm8.*
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB *xmm1, xmm2, r32/m8, imm8*	RVMI	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.NDS.128.66.0F3A.W0 22 /r ib VPINSRD *xmm1, xmm2, r/m32, imm8*	RVMI	V/V	AVX	Insert a dword integer value from *r32/m32* and rest from *xmm2* into *xmm1* at the dword offset in *imm8.*
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ *xmm1, xmm2, r/m64, imm8*	RVMI	V/I	AVX	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).

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

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 an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

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

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #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 *32)) AND MASK) ;
ESAC;
        DEST ← ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX.128 encoded version)

```
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

VPINSRD (VEX.128 encoded version)

```
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

VPINSRQ (VEX.128 encoded version)

```
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[VLMAX-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

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VPINSRQ in non-64-bit mode with VEX.W=1.

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C4 /*r* ib[1] PINSRW *mm, r32/m16, imm8*	RMI	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*	RMI	V/V	SSE2	Move the low word of *r32* or from *m16* into *xmm* at the word position specified by *imm8*.
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW *xmm1, xmm2, r32/m16, imm8*	RVMI	V[2]/V	AVX	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, "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	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

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

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

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

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

```
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[VLMAX-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

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VPINSRW in non-64-bit mode with VEX.W=1.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 04 /r[1] PMADDUBSW *mm1, mm2/m64*	RM	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*	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *ymm1*.

NOTES:

1. See note in Section 2.4, "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)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

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

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

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

VEX.256 encoded version: 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

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[VLMAX-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])
```

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F5 /r[1] PMADDWD *mm, mm/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG F5 /r VPMADDWD *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG F5 /r VPMADDWD *ymm1, ymm2, ymm3/m256*	RVM	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*.

NOTES:

1. See note in Section 2.4, "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

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 4-7 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, 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

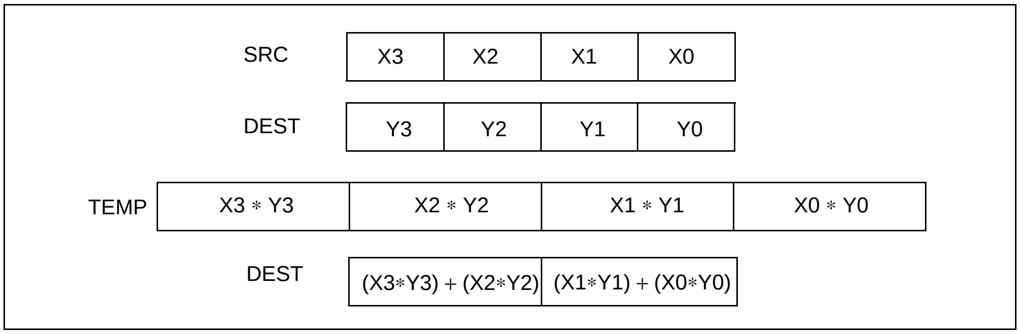

Figure 4-7. PMADDWD Execution Model Using 64-bit Operands

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[VLMAX-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])
```

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMAXSB — Maximum of Packed Signed Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3C /r PMAXSB *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed signed byte integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* and store packed maximum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m128* and store packed maximum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[7:0] > SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; FI;
IF (DEST[15:8] > SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; FI;
IF (DEST[23:16] > SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; FI;
IF (DEST[31:24] > SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; FI;
IF (DEST[39:32] > SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; FI;
IF (DEST[47:40] > SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];
```

```
    ELSE DEST[47:40] ← SRC[47:40]; FI;
IF (DEST[55:48] > SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; FI;
IF (DEST[63:56] > SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; FI;
IF (DEST[71:64] > SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; FI;
IF (DEST[79:72] > SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; FI;
IF (DEST[87:80] > SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; FI;
IF (DEST[95:88] > SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; FI;
IF (DEST[103:96] > SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; FI;
IF (DEST[111:104] > SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; FI;
IF (DEST[119:112] > SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; FI;
IF (DEST[127:120] > SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; FI;
```

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[VLMAX-1:128] ← 0
```

VPMAXSB (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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMAXSB: __m128i _mm_max_epi8 (__m128i a, __m128i b);

VPMAXSB: __m256i _mm256_max_epi8 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXSD — Maximum of Packed Signed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3D /r PMAXSD *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed signed dword integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed dword integers in *xmm2* and *xmm3/m128* and store packed maximum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed dword integers in *ymm2* and *ymm3/m128* and store packed maximum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[127:95] > SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSD: __m128i _mm_max_epi32 (__m128i a, __m128i b);

VPMAXSD: __m256i _mm256_max_epi32 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXSW—Maximum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EE /*r*[1] PMAXSW *mm1, mm2/m64*	RM	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F EE /*r* PMAXSW *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare signed word integers in *xmm2/m128* and *xmm1* and return maximum values.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed word integers in *xmm3/m128* and *xmm2* and store packed maximum values in *xmm1*.
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed word integers in *ymm3/m128* and *ymm2* and store packed maximum values in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

Performs a SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of word integers to the destination operand.

In 64-bit mode, 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 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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;
```

PMAXSW (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;
```

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[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PMAXSW:      __m64 _mm_max_pi16(__m64 a, __m64 b)
(V)PMAXSW:   __m128i _mm_max_epi16 ( __m128i a, __m128i b)
VPMAXSW:     __m256i _mm256_max_epi16 ( __m256i a, __m256i b)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
0F DE /*r*[1] PMAXUB *mm1, mm2/m64*	RM	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /*r* PMAXUB *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns maximum values.
VEX.NDS.128.66.0F.WIG DE /r VPMAXUB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed unsigned byte integers in *xmm2* and *xmm3/m128* and store packed maximum values in *xmm1*.
VEX.NDS.256.66.0F.WIG DE /r VPMAXUB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed unsigned byte integers in *ymm2* and *ymm3/m256* and store packed maximum values in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

Performs a SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of byte integers to the destination operand.

In 64-bit mode, 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 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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 operands)

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

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[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB:	__m64 _mm_max_pu8(__m64 a, __m64 b)
(V)PMAXUB:	__m128i _mm_max_epu8 (__m128i a, __m128i b)
VPMAXUB:	__m256i _mm256_max_epu8 (__m256i a, __m256i b);

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMAXUD — Maximum of Packed Unsigned Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed unsigned dword integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed unsigned dword integers in *xmm2* and *xmm3/m128* and store packed maximum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed unsigned dword integers in *ymm2* and *ymm3/m256* and store packed maximum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[127:95] > SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMAXUD: __m128i _mm_max_epu32 (__m128i a, __m128i b);

VPMAXUD: __m256i _mm256_max_epu32 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXUW — Maximum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3E /r PMAXUW *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed unsigned word integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 3E/r VPMAXUW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed unsigned word integers in *xmm3/m128* and *xmm2* and store maximum packed values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed unsigned word integers in *ymm3/m256* and *ymm2* and store maximum packed values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[15:0] > SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; FI;
IF (DEST[31:16] > SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; FI;
IF (DEST[47:32] > SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; FI;
IF (DEST[63:48] > SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; FI;
IF (DEST[79:64] > SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; FI;
IF (DEST[95:80] > SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];
```

```
    ELSE DEST[95:80] ← SRC[95:80]; FI;
IF (DEST[111:96] > SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; FI;
IF (DEST[127:112] > SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; FI;
```

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[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMAXUW: __m128i _mm_max_epu16 (__m128i a, __m128i b);

VPMAXUW: __m256i _mm256_max_epu16 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMINSB — Minimum of Packed Signed Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 38 /r PMINSB *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed signed byte integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 38 /r VPMINSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* and store packed minimum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 38 /r VPMINSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m256* and store packed minimum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[7:0] < SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; FI;
IF (DEST[15:8] < SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; FI;
IF (DEST[23:16] < SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; FI;
IF (DEST[31:24] < SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; FI;
IF (DEST[39:32] < SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; FI;
IF (DEST[47:40] < SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];
```

```
    ELSE DEST[47:40] ← SRC[47:40]; FI;
IF (DEST[55:48] < SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; FI;
IF (DEST[63:56] < SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; FI;
IF (DEST[71:64] < SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; FI;
IF (DEST[79:72] < SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; FI;
IF (DEST[87:80] < SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; FI;
IF (DEST[95:88] < SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; FI;
IF (DEST[103:96] < SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; FI;
IF (DEST[111:104] < SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; FI;
IF (DEST[119:112] < SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; FI;
IF (DEST[127:120] < SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; FI;
```

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[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINSB: __m128i _mm_min_epi8 (__m128i a, __m128i b);

VPMINSB: __m256i _mm256_min_epi8 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINSD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed signed dword integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed dword integers in *xmm2* and *xmm3/m128* and store packed minimum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed dword integers in *ymm2* and *ymm3/m128* and store packed minimum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[127:95] < SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINSD: __m128i _mm_min_epi32 (__m128i a, __m128i b);

VPMINSD: __m256i _mm256_min_epi32 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMINSW—Minimum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EA /*r*[1] PMINSW *mm1, mm2/m64*	RM	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F EA /*r* PMINSW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Compare signed word integers in *xmm2/m128* and *xmm1* and return minimum values.
VEX.NDS.128.66.0F.WIG EA /r VPMINSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed signed word integers in *xmm3/m128* and *xmm2* and return packed minimum values in *xmm1*.
VEX.NDS.256.66.0F.WIG EA /r VPMINSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed signed word integers in *ymm3/m256* and *ymm2* and return packed minimum values in *ymm1*.

NOTES:

1. See note in Section 2.4, "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

Performs a SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of word integers to the destination operand.

In 64-bit mode, 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 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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;
```

PMINSW (128-bit operands)

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

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[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINSW: __m64 _mm_min_pi16 (__m64 a, __m64 b)

(V)PMINSW: __m128i _mm_min_epi16 (__m128i a, __m128i b)

VPMINSW: __m256i _mm256_min_epi16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DA /*r*[1] PMINUB *mm1, mm2/m64*	RM	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values.
66 0F DA /*r* PMINUB *xmm1, xmm2/m128*	RM	V/V	SSE2	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns minimum values.
VEX.NDS.128.66.0F.WIG DA /r VPMINUB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed unsigned byte integers in *xmm2* and *xmm3/m128* and store packed minimum values in *xmm1*.
VEX.NDS.256.66.0F.WIG DA /r VPMINUB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed unsigned byte integers in *ymm2* and *ymm3/m256* and store packed minimum values in *ymm1*.

NOTES:

1. See note in Section 2.4, “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

Performs a SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of byte integers to the destination operand.

In 64-bit mode, 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 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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 (for 128-bit operands)

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

VPMINUB (VEX.128 encoded version)

```
VPMINUB instruction for 128-bit operands:
    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[VLMAX-1:128] ← 0
```

VPMINUB (VEX.256 encoded version)

```
VPMINUB instruction for 128-bit operands:
    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;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB:	__m64 _m_min_pu8 (__m64 a, __m64 b)
(V)PMINUB:	__m128i _mm_min_epu8 (__m128i a, __m128i b)
VPMINUB:	__m256i _mm256_min_epu8 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMINUD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed unsigned dword integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed unsigned dword integers in *xmm2* and *xmm3/m128* and store packed minimum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed unsigned dword integers in *ymm2* and *ymm3/m256* and store packed minimum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[127:95] < SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX-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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINUD: __m128i _mm_min_epu32 (__m128i a, __m128i b);

VPMINUD: __m256i _mm256_min_epu32 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINUW — Minimum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3A /r PMINUW *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Compare packed unsigned word integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*.
VEX.NDS.128.66.0F38.WIG 3A/r VPMINUW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Compare packed unsigned word integers in *xmm3/m128* and *xmm2* and return packed minimum values in *xmm1*.
VEX.NDS.256.66.0F38.WIG 3A /r VPMINUW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Compare packed unsigned word integers in *ymm3/m256* and *ymm2* and return packed minimum values in *ymm1*.

Instruction Operand Encoding

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

Description

Compares packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[15:0] < SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; FI;
IF (DEST[31:16] < SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; FI;
IF (DEST[47:32] < SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; FI;
IF (DEST[63:48] < SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; FI;
IF (DEST[79:64] < SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; FI;
IF (DEST[95:80] < SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];
```

```
    ELSE DEST[95:80] ← SRC[95:80]; FI;
IF (DEST[111:96] < SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; FI;
IF (DEST[127:112] < SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; FI;
```

VPMINUW (VEX.128 encoded version)

```
VPMINUW instruction for 128-bit operands:
    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[VLMAX-1:128] ← 0
```

VPMINUW (VEX.256 encoded version)

```
VPMINUW instruction for 128-bit operands:
    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;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINUW: __m128i _mm_min_epu16 (__m128i a, __m128i b);

VPMINUW: __m256i _mm256_min_epu16 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMOVMSKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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, "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*	RM	V/V	SSE4_1	Sign extend 8 packed signed 8-bit integers in the low 8 bytes of *xmm2/m64* to 8 packed signed 16-bit integers in *xmm1.*
66 0f 38 21 /r PMOVSXBD *xmm1, xmm2/m32*	RM	V/V	SSE4_1	Sign extend 4 packed signed 8-bit integers in the low 4 bytes of *xmm2/m32* to 4 packed signed 32-bit integers in *xmm1.*
66 0f 38 22 /r PMOVSXBQ *xmm1, xmm2/m16*	RM	V/V	SSE4_1	Sign extend 2 packed signed 8-bit integers in the low 2 bytes of *xmm2/m16* to 2 packed signed 64-bit integers in *xmm1*.
66 0f 38 23 /r PMOVSXWD *xmm1, xmm2/m64*	RM	V/V	SSE4_1	Sign extend 4 packed signed 16-bit integers in the low 8 bytes of *xmm2/m64* to 4 packed signed 32-bit integers in *xmm1*.
66 0f 38 24 /r PMOVSXWQ *xmm1, xmm2/m32*	RM	V/V	SSE4_1	Sign extend 2 packed signed 16-bit integers in the low 4 bytes of *xmm2/m32* to 2 packed signed 64-bit integers in *xmm1*.
66 0f 38 25 /r PMOVSXDQ *xmm1, xmm2/m64*	RM	V/V	SSE4_1	Sign extend 2 packed signed 32-bit integers in the low 8 bytes of *xmm2/m64* to 2 packed signed 64-bit integers in *xmm1*.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW *xmm1, xmm2/m64*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ *ymm1, xmm2/m64*	RM	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*	RM	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*.

Instruction Operand Encoding

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

Description

Sign-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

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

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

VEX.256 encoded version: The destination register is YMM Register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMOVSXBW

```
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]);
```

PMOVSXBD

```
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]);
```

PMOVSXBQ

```
DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);
```

PMOVSXWD

```
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]);
```

PMOVSXWQ

```
DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);
```

PMOVSXDQ

```
DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);
```

VPMOVSXBW (VEX.128 encoded version)

```
Packed_Sign_Extend_BYTE_to_WORD()
DEST[VLMAX-1:128] ← 0
```

VPMOVSXBD (VEX.128 encoded version)

```
Packed_Sign_Extend_BYTE_to_DWORD()
DEST[VLMAX-1:128] ← 0
```

VPMOVSXBQ (VEX.128 encoded version)

```
Packed_Sign_Extend_BYTE_to_QWORD()
DEST[VLMAX-1:128] ← 0
```

VPMOVSXWD (VEX.128 encoded version)

```
Packed_Sign_Extend_WORD_to_DWORD()
DEST[VLMAX-1:128] ← 0
```

VPMOVSXWQ (VEX.128 encoded version)

```
Packed_Sign_Extend_WORD_to_QWORD()
DEST[VLMAX-1:128] ← 0
```

VPMOVSXDQ (VEX.128 encoded version)

```
Packed_Sign_Extend_DWORD_to_QWORD()
DEST[VLMAX-1:128] ← 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])
```

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])
```

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])
```

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])
```

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])
```

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])

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMOVSXBW:	__m128i _mm_ cvtepi8_epi16 (__m128i a);
VPMOVSXBW:	__m256i _mm256_cvtepi8_epi16 (__m128i a);
(V)PMOVSXBD:	__m128i _mm_ cvtepi8_epi32 (__m128i a);
VPMOVSXBD:	__m256i _mm256_cvtepi8_epi32 (__m128i a);
(V)PMOVSXBQ:	__m128i _mm_ cvtepi8_epi64 (__m128i a);
VPMOVSXBQ:	__m256i _mm256_cvtepi8_epi64 (__m128i a);
(V)PMOVSXWD:	__m128i _mm_ cvtepi16_epi32 (__m128i a);
VPMOVSXWD:	__m256i _mm256_cvtepi16_epi32 (__m128i a);
(V)PMOVSXWQ:	__m128i _mm_ cvtepi16_epi64 (__m128i a);
VPMOVSXWQ:	__m256i _mm256_cvtepi16_epi64 (__m128i a);
(V)PMOVSXDQ:	__m128i _mm_ cvtepi32_epi64 (__m128i a);
VPMOVSXDQ:	__m256i _mm256_cvtepi32_epi64 (__m128i a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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*	RM	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.128.66.0F38.WIG 35 /r VPMOVZXDQ *xmm1, xmm2/m64*	RM	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*	RM	V/V	AVX2	Zero extend 16 packed 8-bit integers in the low 16 bytes of *xmm2/m128* to 16 packed 16-bit integers in *ymm1*.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD *ymm1, xmm2/m64*	RM	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*	RM	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*	RM	V/V	AVX2	Zero extend 8 packed 16-bit integers in the low 16 bytes of *xmm2/m128* to 8 packed 32-bit integers in *ymm1*.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ *ymm1, xmm2/m64*	RM	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*	RM	V/V	AVX2	Zero extend 4 packed 32-bit integers in the low 16 bytes of *xmm2/m128* to 4 packed 64-bit integers 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

Zero-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

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

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

VEX.256 encoded version: The destination register is YMM Register.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMOVZXBW

```
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]);
```

PMOVZXBD

```
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]);
```

PMOVZXQB

```
DEST[63:0] ← ZeroExtend(SRC[7:0]);
DEST[127:64] ← ZeroExtend(SRC[15:8]);
```

PMOVZXWD

```
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]);
```

PMOVZXWQ

DEST[63:0] ← ZeroExtend(SRC[15:0]);
DEST[127:64] ← ZeroExtend(SRC[31:16]);

PMOVZXDQ

DEST[63:0] ← ZeroExtend(SRC[31:0]);
DEST[127:64] ← ZeroExtend(SRC[63:32]);

VPMOVZXBW (VEX.128 encoded version)

Packed_Zero_Extend_BYTE_to_WORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXBD (VEX.128 encoded version)

Packed_Zero_Extend_BYTE_to_DWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXBQ (VEX.128 encoded version)

Packed_Zero_Extend_BYTE_to_QWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXWD (VEX.128 encoded version)

Packed_Zero_Extend_WORD_to_DWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXWQ (VEX.128 encoded version)

Packed_Zero_Extend_WORD_to_QWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXDQ (VEX.128 encoded version)

Packed_Zero_Extend_DWORD_to_QWORD()
DEST[VLMAX-1:128] ← 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])

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])

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])

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])

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])

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])

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMOVZXBW:	__m128i _mm_ cvtepu8_epi16 (__m128i a);
VPMOVZXBW:	__m256i _mm256_cvtepu8_epi16 (__m128i a);
(V)PMOVZXBD:	__m128i _mm_ cvtepu8_epi32 (__m128i a);
VPMOVZXBD:	__m256i _mm256_cvtepu8_epi32 (__m128i a);
(V)PMOVZXBQ:	__m128i _mm_ cvtepu8_epi64 (__m128i a);
VPMOVZXBQ:	__m256i _mm256_cvtepu8_epi64 (__m128i a);
(V)PMOVZXWD:	__m128i _mm_ cvtepu16_epi32 (__m128i a);
VPMOVZXWD:	__m256i _mm256_cvtepu16_epi32 (__m128i a);
(V)PMOVZXWQ:	__m128i _mm_ cvtepu16_epi64 (__m128i a);
VPMOVZXWQ:	__m256i _mm256_cvtepu16_epi64 (__m128i a);
(V)PMOVZXDQ:	__m128i _mm_ cvtepu32_epi64 (__m128i a);
VPMOVZXDQ:	__m256i _mm256_cvtepu32_epi64 (__m128i a);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PMULDQ — Multiply Packed Signed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Multiply the packed signed dword integers in *xmm1* and *xmm2/m128* and store the quadword product in *xmm1*.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F38.WIG 28 /r VPMULDQ *ymm1, ymm2, ymm3/m256*	RVM	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*.

Instruction Operand Encoding

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

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand.

For PMULDQ and VPMULDQ (VEX.128 encoded version), the second source operand is two packed signed doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. The first source operand is two packed signed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed signed quadword integers stored in an XMM register. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

For VPMULDQ (VEX.256 encoded version), the second source operand is four packed signed doubleword integers stored in the first (low), third, fifth and seventh doublewords of an YMM register or a 256-bit memory location. The first source operand is four packed signed doubleword integers stored in the first, third, fifth and seventh doublewords of an XMM register. The destination contains four packed signed quadword integers stored in an YMM register. 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.

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).

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 (VLMAX-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 (VLMAX-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.

Operation

PMULDQ (128-bit Legacy SSE version)

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

VPMULDQ (VEX.128 encoded version)

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

VPMULDQ (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]
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULDQ: __m128i _mm_mul_epi32(__m128i a, __m128i b);

VPMULDQ: __m256i _mm256_mul_epi32(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PMULHRSW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 0B /r[1] PMULHRSW *mm1, mm2/m64*	RM	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*	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *ymm1*.

NOTES:

1. See note in Section 2.4, "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

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, use the REX prefix to access additional registers.

Legacy SSE version: 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 (VLMAX-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 (VLMAX-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.

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[VLMAX-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
```

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E4 /r[1] PMULHUW *mm1, mm2/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG E4 /r VPMULHUW *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG E4 /r VPMULHUW *ymm1, ymm2, ymm3/m256*	RVM	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*.

NOTES:

1. See note in Section 2.4, "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

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 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, 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 and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-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.

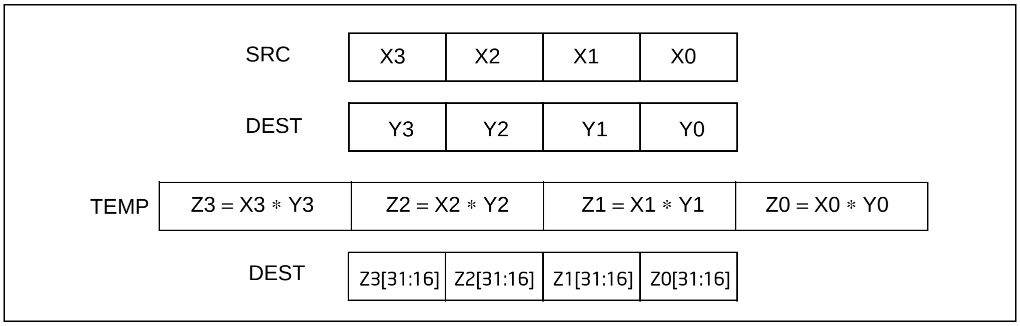

Figure 4-8. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

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[VLMAX-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]

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E5 /r[1] PMULHW *mm, mm/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG E5 /r VPMULHW *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG E5 /r VPMULHW *ymm1, ymm2, ymm3/m256*	RVM	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*.

NOTES:

1. See note in Section 2.4, "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

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 4-8 shows this operation when using 64-bit operands.)

n 64-bit mode, 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 and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-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.

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[VLMAX-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]

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULLD — Multiply Packed Signed Dword 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*	RM	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.NDS.128.66.0F38.WIG 40 /r VPMULLD *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F38.WIG 40 /r VPMULLD *ymm1, ymm2, ymm3/m256*	RVM	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*.

Instruction Operand Encoding

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

Description

Performs four signed multiplications from four pairs of signed dword integers and stores the lower 32 bits of the four 64-bit products in the destination operand (first operand). Each dword element in the destination operand is multiplied with the corresponding dword element of the source operand (second operand) to obtain a 64-bit intermediate product.

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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
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];
```

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[VLMAX-1:128] ← 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]

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULLUD: __m128i _mm_mullo_epi32(__m128i a, __m128i b);

VPMULLD: __m256i _mm256_mullo_epi32(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D5 /r[1] PMULLW *mm, mm/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG D5 /r VPMULLW *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG D5 /r VPMULLW *ymm1, ymm2, ymm3/m256*	RVM	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*.

NOTES:

1. See note in Section 2.4, "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

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 4-8 shows this operation when using 64-bit operands.)

In 64-bit mode, 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 and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-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.

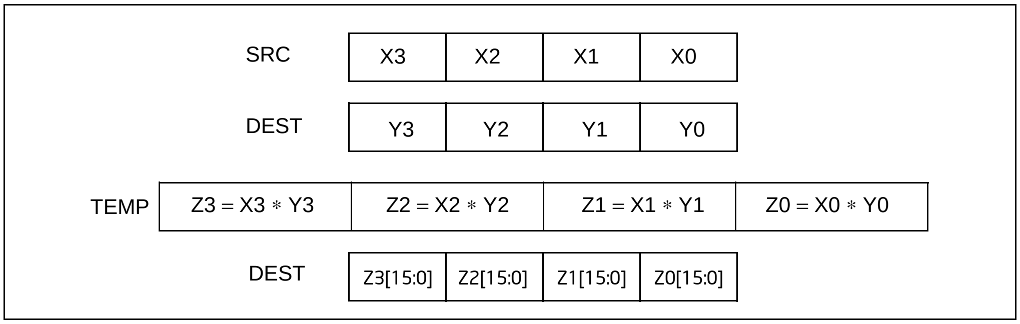

Figure 4-9. PMULLU Instruction Operation Using 64-bit Operands

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];
```

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[VLMAX-1:128] ← 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]
```

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F4 /r[1] PMULUDQ *mm1, mm2/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG F4 /r VPMULUDQ *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG F4 /r VPMULUDQ *ymm1, ymm2, ymm3/m256*	RVM	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*.

NOTES:

1. See note in Section 2.4, "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

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

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

Legacy SSE version: 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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[VLMAX-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]
```

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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	NP	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	NP	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	NP	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	NP	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	NP	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	NP	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	NP	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	NP	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	NP	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
NP	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.

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

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 *)
```

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

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

   POP SS
   POP SS
   POP ESP

```
                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(U)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#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	NP	Invalid	Valid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	NP	Invalid	Valid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	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	NP	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	NP	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPFQ	NP	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
NP	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 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 the Table 4-12 and 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[1], 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.

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

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

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 4-12. 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
	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

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 VM = 0 (* Not in Virtual-8086 Mode *)
    THEN IF CPL = 0
        THEN
            IF OperandSize = 32;
                THEN
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                    VIP, 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 IF CR4.VME = 1 (* In Virtual-8086 Mode with VME Enabled *)
    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)
            THEN
                #GP(0);  (* Trap to virtual-8086 monitor. *)
            ELSE (* Operandsize = 16 *)
                tempFLAGS ← Pop();
                IF EFLAGS.VIP = 1 AND tempFLAGS[9] = 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;
  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. *)
    ELSE (* IOPL < 3 *)
        #GP(0); (* Trap to virtual-8086 monitor. *)
    FI;
  FI;
FI;
```

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#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 the I/O privilege level is less than 3.
	If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#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
0F EB /*r*[1] POR *mm, mm/m64*	RM	V/V	MMX	Bitwise OR of *mm/m64* and *mm*.
66 0F EB /*r* POR *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Bitwise OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EB /r VPOR *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise OR of *xmm2/m128* and *xmm3*.
VEX.NDS.256.66.0F.WIG EB /r VPOR *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Bitwise OR of *ymm2/m256* and *ymm3*.

NOTES:

1. See note in Section 2.4, "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

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, 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 (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

POR (128-bit Legacy SSE version)

DEST ← DEST OR SRC
DEST[VLMAX-1:128] (Unmodified)

VPOR (VEX.128 encoded version)

DEST ← SRC1 OR SRC2
DEST[VLMAX-1:128] ← 0

VPOR (VEX.256 encoded version)

DEST ← SRC1 OR SRC2

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

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

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

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

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

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

Operation

```
FETCH (m8);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch(char *p, int i)
```

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

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#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	PRFCHW	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 all 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 any other cached copy 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.

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 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.

PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /2 PREFETCHWT1 m8	M	V/V	PREFETCHWT1	Move data from m8 closer to the processor using T1 hint with intent to 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 line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

- T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

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

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

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

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

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

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

Operation

```
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request
for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a
hint for the processor and may be skipped depending on implementation.

Prefetch (m8, Level = 1, EXCLUSIVE=1);
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch( char const *, int hint= _MM_HINT_ET1);
```

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
0F F6 /r[1] PSADBW *mm1, mm2/m64*	RM	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*	RM	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.NDS.128.66.0F.WIG F6 /r VPSADBW *xmm1, xmm2, xmm3/m128*	RVM	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.NDS.256.66.0F.WIG F6 /r VPSADBW *ymm1, ymm2, ymm3/m256*	RVM	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.

NOTES:

1. See note in Section 2.4, “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

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 4-10 shows the operation of the PSADBW instruction when using 64-bit operands.

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

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

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

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

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

VEX.256 encoded version: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

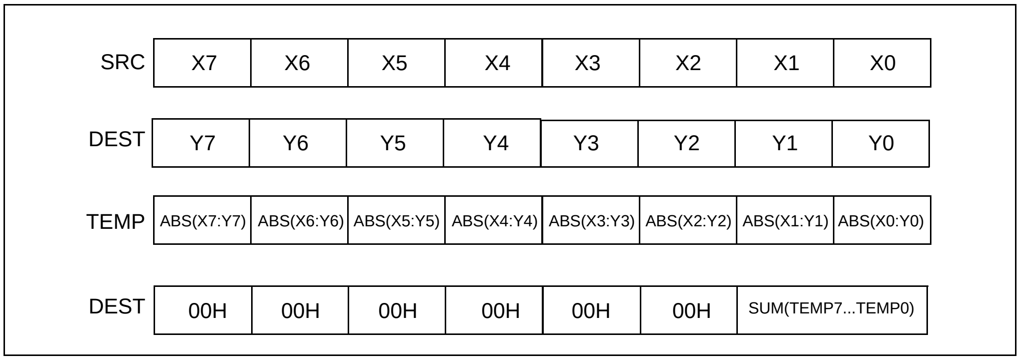

Figure 4-10. PSADBW Instruction Operation Using 64-bit Operands

Operation

PSADBW (when using 64-bit operands)

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

PSADBW (when using 128-bit operands)

```
    TEMP0 ← ABS(DEST[7:0] − SRC[7:0]);
    (* Repeat operation for bytes 2 through 14 *)
    TEMP15 ← ABS(DEST[127:120] − SRC[127:120]);
    DEST[15:0] ← SUM(TEMP0:TEMP7);
    DEST[63:16] ← 000000000000H;
    DEST[79:64] ← SUM(TEMP8:TEMP15);
    DEST[127:80] ← 000000000000H;
    DEST[VLMAX-1:128] (Unmodified)
```

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] ← 00000000000
DEST[VLMAX-1:128] ← 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

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 00 /r[1] PSHUFB *mm1, mm2/m64*	RM	V/V	SSSE3	Shuffle bytes in *mm1* according to contents of *mm2/m64*.
66 0F 38 00 /r PSHUFB *xmm1, xmm2/m128*	RM	V/V	SSSE3	Shuffle bytes in *xmm1* according to contents of *xmm2/m128*.
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shuffle bytes in *xmm2* according to contents of *xmm3/m128*.
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shuffle bytes in *ymm2* according to contents of *ymm3/m256*.

NOTES:

1. See note in Section 2.4, "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

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, use the REX prefix to access additional registers.

Legacy SSE version: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-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 (VLMAX-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.

Note: VEX.L must be 0, otherwise the instruction will #UD.

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
}
DEST[VLMAX-1:128] ← 0
```

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[VLMAX-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
}
```

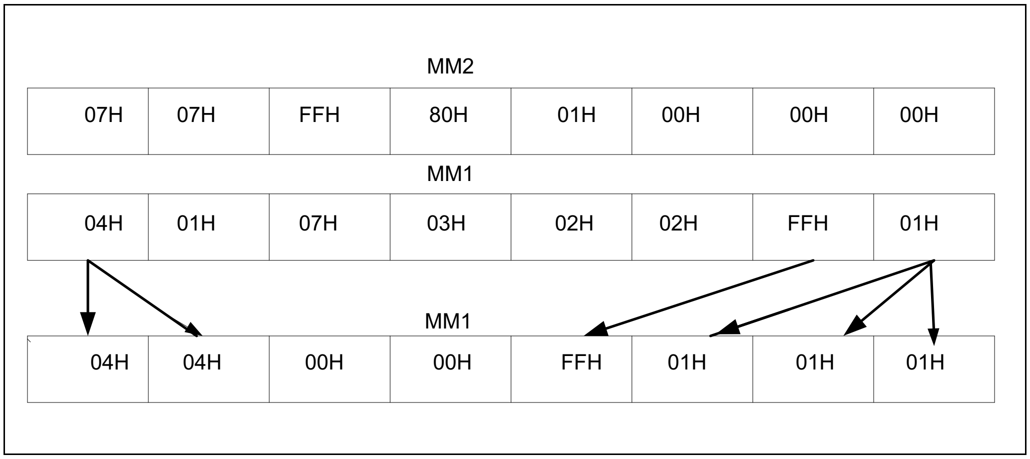

Figure 4-11. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

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*	RMI	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*	RMI	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*	RMI	V/V	AVX2	Shuffle the doublewords in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.

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 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 4-12 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 4-12) 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.

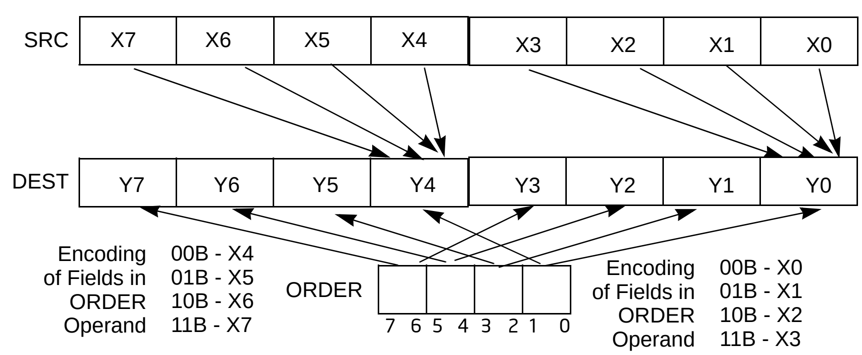

Figure 4-12. 256-bit VPSHUFD Instruction Operation

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.

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

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

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

VEX.256 encoded version: Bits (255:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction 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[VLMAX-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[VLMAX-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];
```

Intel C/C++ Compiler Intrinsic Equivalent

(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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.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*	RMI	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*	RMI	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*	RMI	V/V	AVX2	Shuffle the high words in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.

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 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 4-12. 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, 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 (VLMAX-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 (VLMAX-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.

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[VLMAX-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[VLMAX-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]

Intel C/C++ Compiler Intrinsic Equivalent

(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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.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*	RMI	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*	RMI	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*	RMI	V/V	AVX2	Shuffle the low words in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.

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 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 4-12. 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, 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 (VLMAX-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 (VLMAX-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.

Note: VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, 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[VLMAX-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[VLMAX-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]

Intel C/C++ Compiler Intrinsic Equivalent

(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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PSHUFW—Shuffle Packed Words

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
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 4-12. 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
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.*
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.*
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.NDS.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.NDS.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.NDS.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.NDS.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.NDS.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.NDS.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, "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 (VLMAX-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 (VLMAX-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[VLMAX-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[VLMAX-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[VLMAX-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*	MI	V/V	SSE2	Shift *xmm1* left by *imm8* bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift *xmm2* left by *imm8* bytes while shifting in 0s and store result in *xmm1*.
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift *ymm2* left by *imm8* bytes while shifting in 0s and store result in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	imm8	NA	NA
VMI	VEX.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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-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.

Note: VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSLLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSLLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

VPSLLDQ (VEX.256 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP fl 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[255:128] ← SRC[255:128] << (TEMP * 8)
```

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)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F1 /*r*[1] PSLLW *mm, mm/m64*	RM	V/V	MMX	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /*r* PSLLW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 71 /6 ib PSLLW *mm1*, *imm8*	MI	V/V	MMX	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib PSLLW *xmm1*, *imm8*	MI	V/V	SSE2	Shift words in *xmm1* left by *imm8* while shifting in 0s.
0F F2 /*r*[1] PSLLD *mm, mm/m64*	RM	V/V	MMX	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /*r* PSLLD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 72 /6 ib[1] PSLLD *mm, imm8*	MI	V/V	MMX	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib PSLLD *xmm1*, *imm8*	MI	V/V	SSE2	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
0F F3 /*r*[1] PSLLQ *mm, mm/m64*	RM	V/V	MMX	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /*r* PSLLQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 73 /6 ib[1] PSLLQ *mm, imm8*	MI	V/V	MMX	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib PSLLQ *xmm1*, *imm8*	MI	V/V	SSE2	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift words in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift words in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift doublewords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift doublewords in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift quadwords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift quadwords in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift words in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift words in *ymm2* left by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift doublewords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift doublewords in *ymm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift quadwords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift quadwords in *ymm2* left by *imm8* while shifting in 0s.

NOTES:

1. See note in Section 2.4, "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
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	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 4-13 gives an example of shifting words in a 64-bit operand.

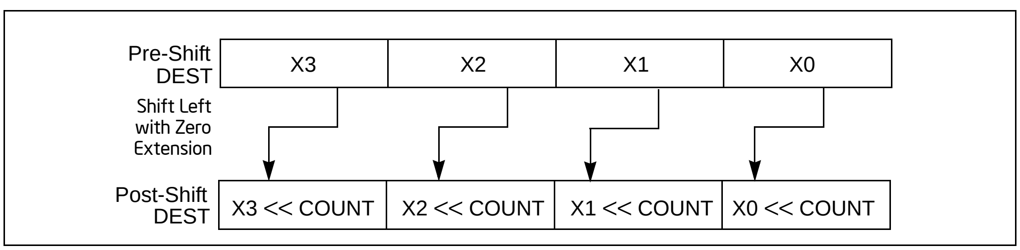

Figure 4-13. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

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

Legacy SSE instructions: 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 (VLMAX-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 (VLMAX-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 and first source operands are YMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

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;
```

PSLLW (with 128-bit operand)

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

PSLLD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
ELSE
    DEST[31:0]  ← ZeroExtend(DEST[31:0] << COUNT);
```

```
        (* Repeat shift operation for 2nd and 3rd doublewords *)
        DEST[127:96] ← ZeroExtend(DEST[127:96] << COUNT);
    FI;
```

PSLLQ (with 128-bit operand)

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

PSLLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLW (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLD (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLD (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSLLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLW (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSLLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLW (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

VPSLLW (ymm, ymm, xmm/m128)
DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)

VPSLLW (ymm, imm8)
DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORD_256bS(SRC1, imm8)

VPSLLD (ymm, ymm, xmm/m128)
DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)

VPSLLD (ymm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)

VPSLLQ (ymm, ymm, xmm/m128)
DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)

VPSLLQ (ymm, imm8)
DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)

Intel C/C++ Compiler Intrinsic Equivalents

PSLLW:	__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW:	__m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW:	__m128i _mm_slli_pi16(__m64 m, int count)
(V)PSLLW:	__m128i _mm_slli_pi16(__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

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally

#UD If VEX.L = 1.

PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E1 /*r*[1] PSRAW *mm, mm/m64*	RM	V/V	MMX	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /*r* PSRAW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
0F 71 /4 ib[1] PSRAW *mm, imm8*	MI	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib PSRAW *xmm1*, imm8	MI	V/V	SSE2	Shift words in *xmm1* right by imm8 while shifting in sign bits
0F E2 /*r*[1] PSRAD *mm, mm/m64*	RM	V/V	MMX	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /*r* PSRAD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
0F 72 /4 ib[1] PSRAD *mm, imm8*	MI	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib PSRAD *xmm1*, imm8	MI	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in sign bits.

NOTES:

1. See note in Section 2.4, “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
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA

Description

Shifts the bits in the individual data elements (words or doublewords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-14 gives an example of shifting words in a 64-bit operand.)

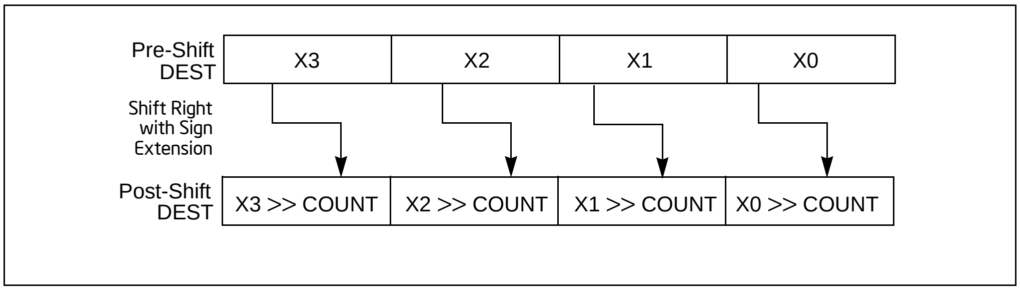

Figure 4-14. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

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

Legacy SSE instructions: 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 (VLMAX-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 (VLMAX-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 and first source operands are YMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

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);
```

PSRAW (with 128-bit operand)

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

PSRAD (with 128-bit operand)

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

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0
```

PSRAD (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRAD (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSRAD (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRAD (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

VPSRAW (ymm, ymm, xmm/m128)
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)

VPSRAW (ymm, imm8)
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)

VPSRAD (ymm, ymm, xmm/m128)
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)

VPSRAD (ymm, imm8)
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally

#UD	If VEX.L = 1.

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*	MI	V/V	SSE2	Shift *xmm1* right by *imm8* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift *xmm2* right by *imm8* bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift *ymm1* right by *imm8* bytes while shifting in 0s.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	imm8	NA	NA
VMI	VEX.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, 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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-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.

Note: VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSRLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSRLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

VPSRLDQ (VEX.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)
```

Intel C/C++ Compiler Intrinsic Equivalents

(V)PSRLDQ: __m128i _mm_srli_si128 (__m128i a, int imm)

VPSRLDQ: __m256i _mm256_srli_si256 (__m256i a, const int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D1 /*r*[1] PSRLW *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 71 /2 ib[1] PSRLW *mm, imm8*	MI	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib PSRLW *xmm1, imm8*	MI	V/V	SSE2	Shift words in *xmm1* right by *imm8* while shifting in 0s.
0F D2 /*r*[1] PSRLD *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
0F 72 /2 ib[1] PSRLD *mm, imm8*	MI	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib PSRLD *xmm1*, imm8	MI	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
0F D3 /*r*[1] PSRLQ *mm, mm/m64*	RM	V/V	MMX	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /*r* PSRLQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 73 /2 ib[1] PSRLQ *mm, imm8*	MI	V/V	MMX	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib PSRLQ *xmm1, imm8*	MI	V/V	SSE2	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Shift quadwords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ *xmm1, xmm2, imm8*	VMI	V/V	AVX	Shift quadwords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ *ymm1, ymm2, xmm3/m128*	RVM	V/V	AVX2	Shift quadwords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ *ymm1, ymm2, imm8*	VMI	V/V	AVX2	Shift quadwords in *ymm2* right by *imm8* while shifting in 0s.

NOTES:

1. See note in Section 2.4, "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
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	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 4-15 gives an example of shifting words in a 64-bit operand.

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

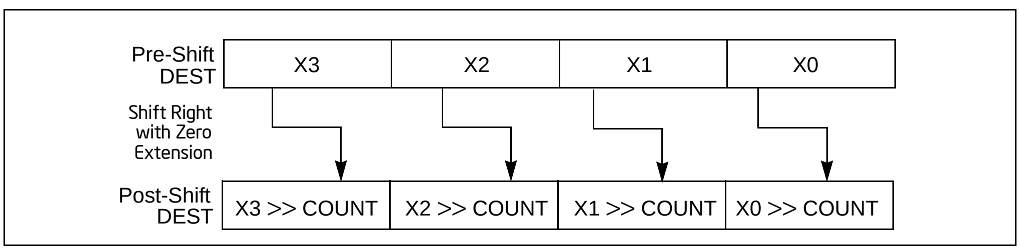

Figure 4-15. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

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

Legacy SSE instructions: 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 (VLMAX-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 (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 128-bit memory location or an 8-bit immediate.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

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;
```

PSRLW (with 128-bit operand)

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

PSRLD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H
```

```
    ELSE
        DEST[31:0]  ← ZeroExtend(DEST[31:0] >> COUNT);
        (* Repeat shift operation for 2nd and 3rd doublewords *)
        DEST[127:96] ← ZeroExtend(DEST[127:96] >> COUNT);
    FI;
```

PSRLQ (with 128-bit operand)

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

PSRLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRLW (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSRLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRLW (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSRLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRLD (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSRLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRLD (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSRLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)

DEST[VLMAX-1:128] (Unmodified)

VPSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

VPSRLW (ymm, ymm, xmm/m128)
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)

VPSRLW (ymm, imm8)
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)

VPSRLD (ymm, ymm, xmm/m128)
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)

VPSRLD (ymm, imm8)
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)

VPSRLQ (ymm, ymm, xmm/m128)
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)

VPSRLQ (ymm, imm8)
DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4 and 7 for non-VEX-encoded instructions; additionally

#UD	If VEX.L = 1.

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F8 /r[1] PSUBB *mm, mm/m64*	RM	V/V	MMX	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*.
66 0F F8 /r PSUBB *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
0F F9 /r[1] PSUBW *mm, mm/m64*	RM	V/V	MMX	Subtract packed word integers in *mm/m64* from packed word integers in *mm*.
66 0F F9 /r PSUBW *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
0F FA /r[1] PSUBD *mm, mm/m64*	RM	V/V	MMX	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*.
66 0F FA /r PSUBD *xmm1, xmm2/m128*	RM	V/V	SSE2	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed byte integers in *xmm3/m128* from *xmm2*.
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed word integers in *xmm3/m128* from *xmm2*.
VEX.NDS.128.66.0F.WIG FA /r VPSUBD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed doubleword integers in *xmm3/m128* from *xmm2*.
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed byte integers in *ymm3/m256* from *ymm2*.
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed word integers in *ymm3/m256* from *ymm2*.
VEX.NDS.256.66.0F.WIG FA /r VPSUBD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed doubleword integers in *ymm3/m256* from *ymm2*.

NOTES:

1. See note in Section 2.4, "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

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

Legacy SSE version: When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

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];
```

PSUBB (with 128-bit operands)

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

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[VLMAX-1:128] ← 00
```

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]
```

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];
```

PSUBW (with 128-bit operands)

```
    DEST[15:0]  ← DEST[15:0] − SRC[15:0];
    (* Repeat subtract operation for 2nd through 7th word *)
    DEST[127:112] ← DEST[127:112] − SRC[127:112];
```

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[VLMAX-1:128] ← 0
```

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]
```

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];
```

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[VLMAX-1:128] ← 0
```

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]
```

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FB /r[1] PSUBQ *mm1*, *mm2/m64*	RM	V/V	SSE2	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /r PSUBQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed quadword integers in *xmm3/m128* from *xmm2*.
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed quadword integers in *ymm3/m256* from *ymm2*.

NOTES:

1. See note in Section 2.4, "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

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, 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 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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

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[VLMAX-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]
```

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E8 /*r*[1] PSUBSB *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
0F E9 /*r*[1] PSUBSW *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed signed byte integers in *xmm3/m128* from packed signed byte integers in *xmm2* and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed signed word integers in *xmm3/m128* from packed signed word integers in *xmm2* and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed signed byte integers in *ymm3/m256* from packed signed byte integers in *ymm2* and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed signed word integers in *ymm3/m256* from packed signed word integers in *ymm2* and saturate results.

NOTES:

1. See note in Section 2.4, "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

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

Legacy SSE version: When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

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] );
```

PSUBSB (with 128-bit operands)

```
    DEST[7:0] ← SaturateToSignedByte (DEST[7:0] − SRC[7:0]);
    (* Repeat subtract operation for 2nd through 14th bytes *)
    DEST[127:120] ← SaturateToSignedByte (DEST[127:120] − SRC[127:120]);
```

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[VLMAX-1:128] ← 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]);
```

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] );
```

PSUBSW (with 128-bit operands)

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

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[VLMAX-1:128] ← 0
```

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]);

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D8 /r[1] PSUBUSB *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
0F D9 /r[1] PSUBUSW *mm, mm/m64*	RM	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*	RM	V/V	SSE2	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in *xmm1* and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed unsigned byte integers in *xmm3/m128* from packed unsigned byte integers in *xmm2* and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed unsigned word integers in *xmm3/m128* from packed unsigned word integers in *xmm2* and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed unsigned byte integers in *ymm3/m256* from packed unsigned byte integers in *ymm2* and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract packed unsigned word integers in *ymm3/m256* from packed unsigned word integers in *ymm2* and saturate result.

NOTES:

1. See note in Section 2.4, "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

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

Legacy SSE version: When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

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];
```

PSUBUSB (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]);
```

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[VLMAX-1:128] ← 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]);
```

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] );
```

PSUBUSW (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]);
```

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[VLMAX-1:128] ← 0
```

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]);
```

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 68 /*r*[1] PUNPCKHBW *mm, mm/m64*	RM	V/V	MMX	Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm*.
66 0F 68 /*r* PUNPCKHBW *xmm1, xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 69 /*r*[1] PUNPCKHWD *mm, mm/m64*	RM	V/V	MMX	Unpack and interleave high-order words from *mm* and *mm/m64* into *mm*.
66 0F 69 /*r* PUNPCKHWD *xmm1, xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 6A /*r*[1] PUNPCKHDQ *mm, mm/m64*	RM	V/V	MMX	Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm*.
66 0F 6A /*r* PUNPCKHDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1*.
66 0F 6D /*r* PUNPCKHQDQ *xmm1, xmm2/m128*	RM	V/V	SSE2	Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 68/r VPUNPCKHBW *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order bytes from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 69/r VPUNPCKHWD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order words from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 6A/r VPUNPCKHDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order doublewords from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 6D/r VPUNPCKHQDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave high-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6A /r VPUNPCKHDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6D /r VPUNPCKHQDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave high-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.

NOTES:

1. See note in Section 2.4, "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

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 4-16 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

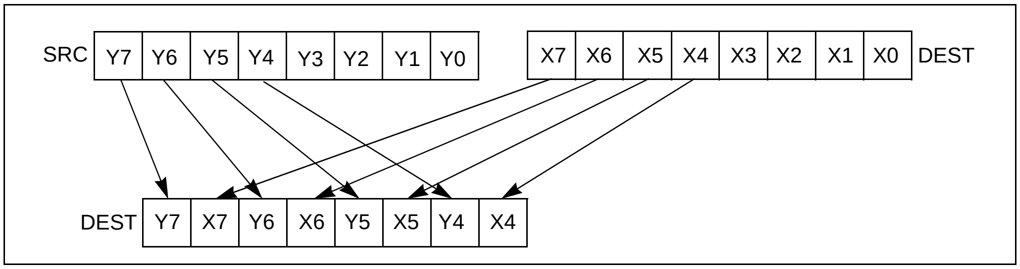

Figure 4-16. PUNPCKHBW Instruction Operation Using 64-bit Operands

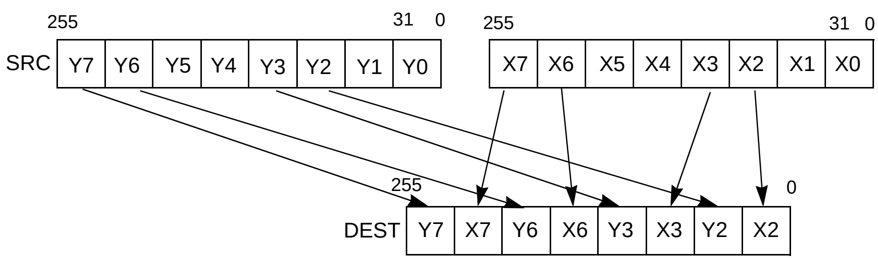

Figure 4-17. 256-bit VPUNPCKHDQ Instruction Operation

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

Legacy SSE versions: 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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PUNPCKHBW instruction with 64-bit operands:

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

PUNPCKHW instruction with 64-bit operands:

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

PUNPCKHDQ instruction with 64-bit operands:

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

PUNPCKHBW instruction with 128-bit operands:

```
DEST[7:0]← DEST[71:64];
DEST[15:8] ← SRC[71:64];
DEST[23:16] ← DEST[79:72];
DEST[31:24] ← SRC[79:72];
DEST[39:32] ← DEST[87:80];
DEST[47:40] ← SRC[87:80];
DEST[55:48] ← DEST[95:88];
DEST[63:56] ← SRC[95:88];
DEST[71:64] ← DEST[103:96];
DEST[79:72] ← SRC[103:96];
DEST[87:80] ← DEST[111:104];
DEST[95:88] ← SRC[111:104];
DEST[103:96] ← DEST[119:112];
DEST[111:104] ← SRC[119:112];
DEST[119:112] ← DEST[127:120];
DEST[127:120] ← SRC[127:120];
```

PUNPCKHWD instruction with 128-bit operands:

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

DEST[127:112] ← SRC[127:112];

PUNPCKHDQ instruction with 128-bit operands:

DEST[31:0] ← DEST[95:64];
DEST[63:32] ← SRC[95:64];
DEST[95:64] ← DEST[127:96];
DEST[127:96] ← SRC[127:96];

PUNPCKHQDQ instruction:

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

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

VPUNPCKHBW (VEX.128 encoded version)
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKHBW (VEX.256 encoded version)
DEST[255:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)

PUNPCKHWD (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPUNPCKHWD (VEX.128 encoded version)
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKHWD (VEX.256 encoded version)
DEST[255:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)

PUNPCKHDQ (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPUNPCKHDQ (VEX.128 encoded version)
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKHDQ (VEX.256 encoded version)
DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)

PUNPCKHQDQ (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHQDQ (VEX.128 encoded version)
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[255:127] ← 0

VPUNPCKHQDQ (VEX.256 encoded version)
DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256(SRC1, SRC2)

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
0F 60 /*r*[1] PUNPCKLBW *mm, mm/m32*	RM	V/V	MMX	Interleave low-order bytes from *mm* and *mm/m32* into *mm.*
66 0F 60 /*r* PUNPCKLBW *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 61 /*r*[1] PUNPCKLWD *mm, mm/m32*	RM	V/V	MMX	Interleave low-order words from *mm* and *mm/m32* into *mm.*
66 0F 61 /*r* PUNPCKLWD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 62 /*r*[1] PUNPCKLDQ *mm, mm/m32*	RM	V/V	MMX	Interleave low-order doublewords from *mm* and *mm/m32* into *mm.*
66 0F 62 /*r* PUNPCKLDQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1.
66 0F 6C /*r* PUNPCKLQDQ *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order bytes from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order words from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order doublewords from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Interleave low-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 62 /r VPUNPCKLDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6C /r VPUNPCKLQDQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Interleave low-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.

NOTES:

1. See note in Section 2.4, "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

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 4-18 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

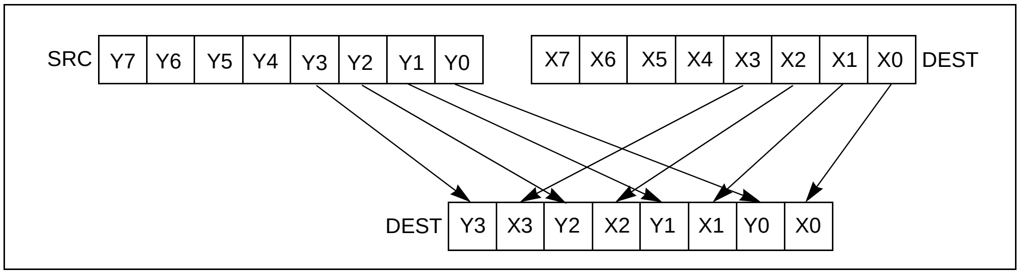

Figure 4-18. PUNPCKLBW Instruction Operation Using 64-bit Operands

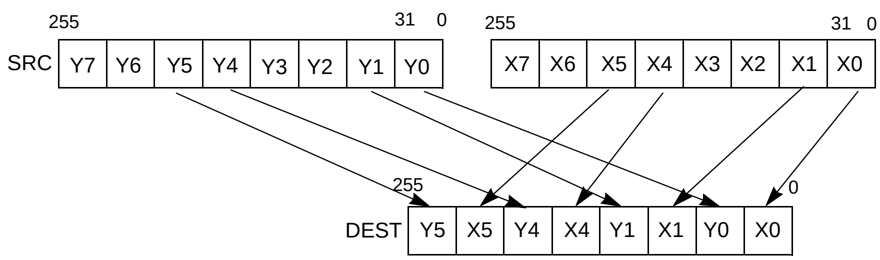

Figure 4-19. 256-bit VPUNPCKLDQ Instruction Operation

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

Legacy SSE versions: 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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PUNPCKLBW instruction with 64-bit operands:

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

PUNPCKLWD instruction with 64-bit operands:

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

PUNPCKLDQ instruction with 64-bit operands:

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

PUNPCKLBW instruction with 128-bit operands:

```
DEST[7:0]← DEST[7:0];
DEST[15:8]  ← SRC[7:0];
DEST[23:16] ← DEST[15:8];
DEST[31:24] ← SRC[15:8];
DEST[39:32] ← DEST[23:16];
DEST[47:40] ← SRC[23:16];
DEST[55:48] ← DEST[31:24];
DEST[63:56] ← SRC[31:24];
DEST[71:64] ← DEST[39:32];
DEST[79:72] ← SRC[39:32];
DEST[87:80] ← DEST[47:40];
DEST[95:88] ← SRC[47:40];
DEST[103:96]  ← DEST[55:48];
DEST[111:104] ← SRC[55:48];
DEST[119:112] ← DEST[63:56];
DEST[127:120] ← SRC[63:56];
```

PUNPCKLWD instruction with 128-bit operands:

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

PUNPCKLDQ instruction with 128-bit operands:

DEST[31:0] ← DEST[31:0];
DEST[63:32] ← SRC[31:0];
DEST[95:64] ← DEST[63:32];
DEST[127:96] ← SRC[63:32];

PUNPCKLQDQ

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

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_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_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_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 (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_BYTES(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKLBW (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_BYTES_128b(SRC1, SRC2)

PUNPCKLWD (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLWD (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKLWD (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_WORDS(SRC1, SRC2)

PUNPCKLDQ (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLDQ (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKLDQ (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)

PUNPCKLQDQ (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLQDQ (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPUNPCKLQDQ (VEX.256 encoded instruction)
DEST[255:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)

Intel C/C++ Compiler Intrinsic Equivalents

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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	NP	Invalid	Valid	Push CS.
16	PUSH SS	NP	Invalid	Valid	Push SS.
1E	PUSH DS	NP	Invalid	Valid	Push DS.
06	PUSH ES	NP	Invalid	Valid	Push ES.
0F A0	PUSH FS	NP	Valid	Valid	Push FS.
0F A8	PUSH GS	NP	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
NP	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	NP	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	NP	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
NP	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—Push EFLAGS Register onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	NP	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	NP	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	NP	Valid	N.E.	Push RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	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 in virtual-8086 mode and the I/O privilege level (IOPL) is less than 3, the PUSHF/PUSHFD instruction causes a general protection exception (#GP).

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 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 EFLAG bits are cleared in image stored on the stack *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
        FI;

    ELSE IF 64-bit MODE (* In 64-bit Mode *)
        IF OperandSize = 64
```

```
                THEN
                    push (RFLAGS AND 00000000_00FCFFFFH);
                    (* VM and RF RFLAG bits are cleared in image stored on the stack; *)
                ELSE
                    push (EFLAGS); (* Lower 16 bits only *)
            FI;
    ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
            #GP(0); (* Trap to virtual-8086 monitor *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#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 the current privilege level is 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
0F EF /*r*[1] PXOR *mm, mm/m64*	RM	V/V	MMX	Bitwise XOR of *mm/m64* and *mm*.
66 0F EF /*r* PXOR *xmm1, xmm2/m128*	RM	V/V	SSE2	Bitwise XOR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EF /r VPXOR *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Bitwise XOR of *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG EF /r VPXOR *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Bitwise XOR of *ymm3/m256* and *ymm2*.

NOTES:

1. See note in Section 2.4, "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

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, 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 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 (VLMAX-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 (VLMAX-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 operand and destination operands are YMM registers.

Note: VEX.L must be 0, otherwise instructions will #UD.

Operation

PXOR (128-bit Legacy SSE version)

```
DEST ← DEST XOR SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPXOR (VEX.128 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[VLMAX-1:128] ← 0
```

VPXOR (VEX.256 encoded version)

DEST ← SRC1 XOR SRC2

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

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. In legacy and compatibility mode, the processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except 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 = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* RCR instruction operation *)
IF COUNT = 1
```

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

(* ROL and ROR instructions *)
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;
CF ← LSB(DEST);
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;
CF ← MSB(DEST);
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR MSB − 1(DEST);
    ELSE OF is undefined;
FI;
```

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see "Description" above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#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
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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

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[VLMAX-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[VLMAX-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.NDS.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 (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

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

Operation

RCPSS (128-bit Legacy SSE version)

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

VRCPSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])
```

DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-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.
REX.W + F3 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.
REX.W + F3 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	NP	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
NP	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 35, "Model-Specific Registers (MSRs)," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, 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.

RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
0F 01 EE	RDPKRU	NP	V/V	OSPKE	Reads PKRU into EAX.

Instruction Operand Encoding

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

Description

The RDPKRU instruction loads 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, a general-protection exception (#GP) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

In 64-bit mode, bits 63:32 of RCX are ignored, and RDPKRU clears bits 63:32 of each of RDX and RAX.

Operation

```
IF (ECX = 0)
    THEN
        EAX ← PKRU;
        EDX ← 0;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

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	NP	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
NP	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 selects one of two type of performance counters, specifies the index relative to the base of each counter type, and selects "fast" read mode if supported. The two counter types are :

- General-purpose or special-purpose performance counters: The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Section 30.13, 30.14 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This counter type is selected if ECX[30] is clear.
- Fixed-function performance counter. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

ECX[29:0] specifies the index. 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. The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf.

Table 4-13 lists valid indices of the general-purpose and special-purpose performance counters according to the derived DisplayFamily_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-M" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 4-13. Valid General and Special Purpose Performance Counter Index Range for RDPMC

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH	0, 1	0, 1
Pentium® 4, Intel® Xeon processors	0FH_00H, 0FH_01H, 0FH_02H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium 4, Intel Xeon processors	(0FH_03H, 0FH_04H, 0FH_06H) and (L3 is absent)	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1

Table 4-13. Valid General and Special Purpose Performance Counter Index Range for RDPMC (Contd.)

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel Xeon processors 7100 series with L3	(0FH_06H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™2 Duo processor family, Intel Xeon processor family - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
Intel® Atom™ processor family	06H_1CH	0, 1	0, 1
Intel® Core™i7 processor, Intel Xeon processors 5500 series	06H_1AH, 06H_1EH, 06H_1FH, 06H_2EH	0-3	0, 1, 2, 3

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters. On Intel Xeon processor 7100 series with L3, performance counters with indices 18-25 are also 32-bit counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDMPC with ECX between from 4000_0000H and 4000_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

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.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

Operation

```
(* Intel Core i7 processor family and Intel Xeon processor 3400, 5500 series*)

Most significant counter bit (MSCB) = 47

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[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;

(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series*)

Most significant counter bit (MSCB) = 39

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;

(* P6 family processors and Pentium processor with MMX technology *)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
(* Processors with CPUID family 15 *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
        THEN IF ECX[31] = 0
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                EDX ← PMC(ECX[30:0])[39:32];
        ELSE (* ECX[31] = 1*)
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
        FI;
    ELSE IF (*64-bit Intel Xeon processor with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
```

```
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE IF (*Intel Xeon processor 7100 series with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-16. *)
        GP(0);
    FI;
ELSE  (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-13).
	(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid performance counter index is specified (see Table 4-13).
	(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.
#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 (see Table 4-13).
	(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.
#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 in ECX[30:0] (see Table 4-13).
#UD	If the LOCK prefix is used.

RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
0F C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
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 the F2H or F3H 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
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.
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.
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 the F2H or F3H 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 the F2H or F3H 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 the F2H or F3H 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 the F2H or F3H 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 the F2H or F3H 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	NP	Valid	Valid	Read time-stamp counter into EDX:EAX.

Instruction Operand Encoding

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

Description

Loads 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. If software requires RDTSC to be executed only after all previous instructions have completed locally, it can either use RDTSCP (if the processor supports that instruction) or execute the sequence LFENCE;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.

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	NP	Valid	Valid	Read 64-bit time-stamp counter and 32-bit IA32_TSC_AUX value into EDX:EAX and ECX.

Instruction Operand Encoding

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

Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also loads the IA32_TSC_AUX MSR (address C000_0103H) 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 waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.

The presence of the RDTSCP instruction is indicated by CPUID leaf 80000001H, EDX bit 27. If the bit is set to 1 then RDTSCP is present on the 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;
        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.

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	NP	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS *m8*, DX	NP	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS *m16*, DX	NP	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI.]
F3 6D	REP INS *m32*, DX	NP	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS *r/m32*, DX	NP	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS *m8, m8*	NP	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS *m8, m8*	NP	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS *m16, m16*	NP	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m32, m32*	NP	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS *m64, m64*	NP	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, *r/m8*	NP	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, *r/m8**	NP	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, *r/m16*	NP	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, *r/m32*	NP	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, *r/m32*	NP	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	NP	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS AL	NP	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	NP	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	NP	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS RAX	NP	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS *m8*	NP	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS *m8*	NP	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS *m16*	NP	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS *m32*	NP	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS *m64*	NP	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS *m8, m8*	NP	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS *m8, m8*	NP	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS *m16, m16*	NP	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m32, m32*	NP	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS *m64, m64*	NP	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS *m8*	NP	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS *m8*	NP	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS *m16*	NP	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS *m32*	NP	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*	NP	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS *m8, m8*	NP	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS *m8, m8*	NP	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS *m16, m16*	NP	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m32, m32*	NP	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS *m64, m64*	NP	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS *m8*	NP	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS *m8*	NP	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS *m16*	NP	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m32*	NP	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS *m64*	NP	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
NP	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 4-14.

Table 4-14. 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.

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	NP	Valid	Valid	Near return to calling procedure.
CB	RET	NP	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
NP	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.

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-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

RETURN-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-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 of segment register (ES, FS, GS, and DS)
        DO
            IF segment register points to data or non-conforming code segment
            and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
                THEN SegmentSelector ← 0; (* Segment selector invalid *)
            FI;
        OD;

    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-OUTER-PRIVILEGE-LEVEL;
                ELSE GOTO IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL;
            FI;
    FI;

IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    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-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 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 Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD	If VEX.W = 1.

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 single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-20. 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 4-15 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 (VLMAX-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 (VLMAX-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.

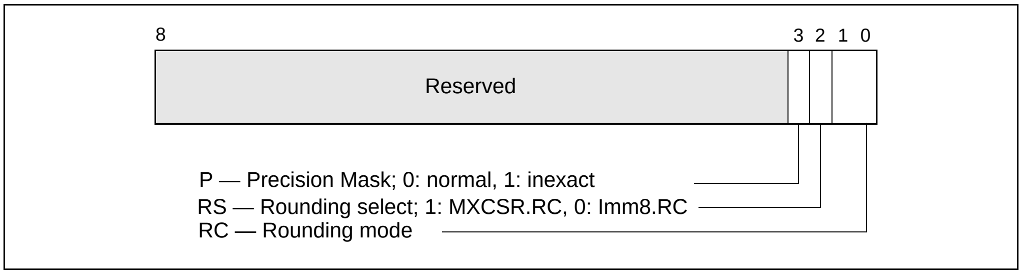

Figure 4-20. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-15. 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[VLMAX-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[VLMAX-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 4-20. 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 4-15 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 (VLMAX-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 (VLMAX-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[VLMAX-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[VLMAX-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.NDS.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 4-20. 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 4-15 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 (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

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

Operation

```
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[VLMAX-1:64] (Unmodified)
```

VROUNDSD (VEX.128 encoded version)

DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-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.NDS.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 4-20. 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 4-15 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 (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

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

Operation

```
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[VLMAX-1:32] (Unmodified)
```

VROUNDSS (VEX.128 encoded version)

DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-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	NP	Invalid	Valid	Resume operation of interrupted program.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	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
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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

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[VLMAX-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[VLMAX-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.NDS.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 (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

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

Operation

RSQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[VLMAX-1:32] (Unmodified)
```

VRSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-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	NP	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
NP	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.NDS[1].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.NDS[1].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.NDS[1].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.NDS[1].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.NDS[1].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.NDS[1].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.*

NOTES:

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

Instruction Operand Encoding

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

Description

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 Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

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*	NP	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.*
AF	SCAS *m16*	NP	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.*
AF	SCAS *m32*	NP	Valid	Valid	Compare EAX with doubleword at ES(E)DI or RDI then set status flags.*
REX.W + AF	SCAS *m64*	NP	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	NP	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.*
AF	SCASW	NP	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.*
AF	SCASD	NP	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCASQ	NP	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
NP	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.

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
0F AE F8	SFENCE	NP	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

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

Description

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

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 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in bytes 3-5, and byte 6 is zero-filled. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In IA-32e mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386™ processors fill these bits with 0s.

Operation

```
IF instruction is SGDT
        IF OperandSize = 16
            THEN
                DEST[0:15] ← GDTR(Limit);
                DEST[16:39] ← GDTR(Base); (* 24 bits of base address stored *)
                DEST[40:47] ← 0;
            ELSE IF (32-bit Operand Size)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)
                FI;
            ELSE (* 64-bit Operand Size *)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

SHLD—Double Precision Shift Left

Opcode*	Instruction	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—Shuffle Packed 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*	RMI	V/V	SSE2	Shuffle packed double-precision floating-point values selected by *imm8* from *xmm1* and *xmm2/m128* to *xmm1*.
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by *imm8* from *xmm2* and *xmm3/mem*.
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX	Shuffle Packed double-precision floating-point values selected by *imm8* from *ymm2* and *ymm3/mem*.

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

Moves either of the two packed double-precision floating-point values from destination operand (first operand) into the low quadword of the destination operand; moves either of the two packed double-precision floating-point values from the source operand into to the high quadword of the destination operand (see Figure 4-21). The select operand (third operand) determines which values are moved to the destination operand.

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

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

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

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

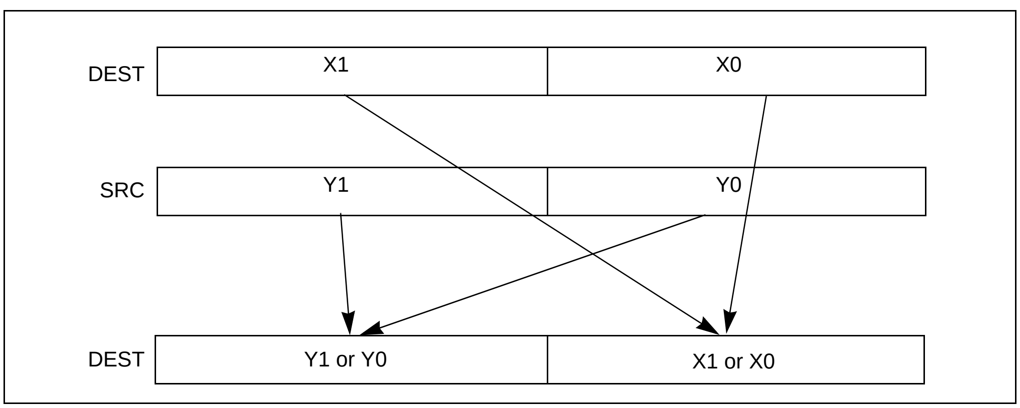

Figure 4-21. SHUFPD Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bit 0 selects which value is moved from the destination operand to the result (where 0 selects the low quadword and 1 selects the high quadword) and bit 1 selects which value is moved from the source operand to the result. Bits 2 through 7 of the select operand are reserved and must be set to 0.

Operation

```
IF SELECT[0] = 0
    THEN DEST[63:0] ← DEST[63:0];
    ELSE DEST[63:0] ← DEST[127:64]; FI;

IF SELECT[1] = 0
    THEN DEST[127:64] ← SRC[63:0];
    ELSE DEST[127:64] ← SRC[127:64]; FI;
```

SHUFPD (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[VLMAX-1:128] (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[VLMAX-1:128] ← 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;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPD: __m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)

VSHUFPD: __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C6 /*r* ib SHUFPS *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE	Shuffle packed single-precision floating-point values selected by *imm8* from *xmm1* and *xmm1/m128* to *xmm1*.
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by *imm8* from *xmm2* and *xmm3/mem*.
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX	Shuffle Packed single-precision floating-point values selected by *imm8* from *ymm2* and *ymm3/mem*.

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

Moves two of the four packed single-precision floating-point values from the destination operand (first operand) into the low quadword of the destination operand; moves two of the four packed single-precision floating-point values from the source operand (second operand) into to the high quadword of the destination operand (see Figure 4-22). The select operand (third operand) determines which values are moved to the destination operand.

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

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

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

determines which values are moved to the destination operand.

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.

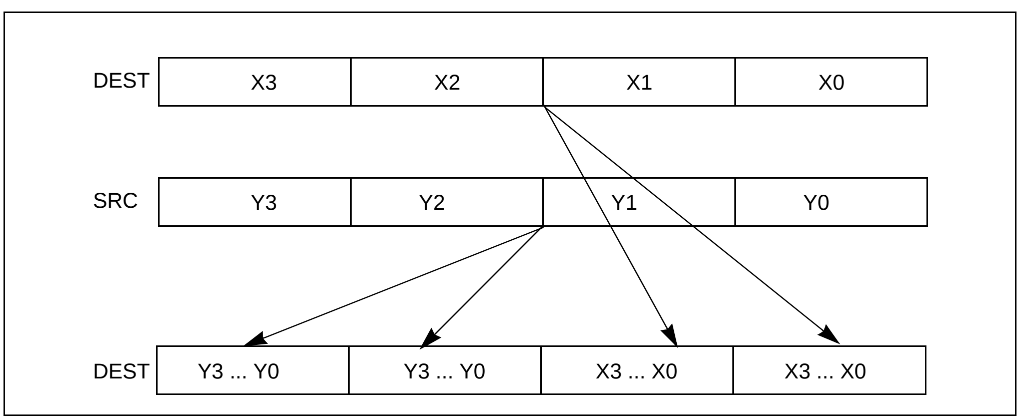

Figure 4-22. SHUFPS Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand to the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand to the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand to the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand to the high doubleword of the result.

Operation

```
CASE (SELECT[1:0]) OF
    0:   DEST[31:0] ← DEST[31:0];
    1:   DEST[31:0] ← DEST[63:32];
    2:   DEST[31:0] ← DEST[95:64];
    3:   DEST[31:0] ← DEST[127:96];
ESAC;

CASE (SELECT[3:2]) OF
    0:   DEST[63:32] ← DEST[31:0];
    1:   DEST[63:32] ← DEST[63:32];
    2:   DEST[63:32] ← DEST[95:64];
    3:   DEST[63:32] ← DEST[127:96];
ESAC;

CASE (SELECT[5:4]) OF
    0:   DEST[95:64] ← SRC[31:0];
    1:   DEST[95:64] ← SRC[63:32];
    2:   DEST[95:64] ← SRC[95:64];
    3:   DEST[95:64] ← SRC[127:96];
ESAC;

CASE (SELECT[7:6]) OF
    0:   DEST[127:96] ← SRC[31:0];
    1:   DEST[127:96] ← SRC[63:32];
    2:   DEST[127:96] ← SRC[95:64];
    3:   DEST[127:96] ← SRC[127:96];
ESAC;
```

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

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[VLMAX-1:128] ← 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]);
```

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPS: __m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)

VSHUFPS: __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

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, if the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

Operation

```
IF instruction is SIDT
    THEN
        IF OperandSize = 16
            THEN
                DEST[0:15] ← IDTR(Limit);
                DEST[16:39] ← IDTR(Base); (* 24 bits of base address stored; *)
                DEST[40:47] ← 0;
            ELSE IF (32-bit Operand Size)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)
            ELSE (* 64-bit Operand Size *)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	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.
#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 SLDT instruction is not recognized in real-address mode.

	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a 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.

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. The is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 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.
#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(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.
#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.

SQRTPD—Compute Square Roots of Packed 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*	RM	V/V	SSE2	Computes square roots of the packed double-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.66.0F.WIG 51 /r VSQRTPD *xmm1, xmm2/m128*	RM	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*	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in *ymm2/m256* and stores the result 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 square roots of the two packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

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

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 (VLMAX-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 (VLMAX-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

SQRTPD (128-bit Legacy SSE version)

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

VSQRTPD (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPD: __m128d _mm_sqrt_pd (m128d a)

SQRTPD: __m256d _mm256_sqrt_pd (__m256d a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv ≠ 1111B.

SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 51 /*r* SQRTPS *xmm1*, *xmm2/m128*	RM	V/V	SSE	Computes square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1.*
VEX.128.0F.WIG 51 /r VSQRTPS *xmm1, xmm2/m128*	RM	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*	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in *ymm2/m256* and stores the result 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 square roots of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

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

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 (VLMAX-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 (VLMAX-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

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

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[VLMAX-1:128] ← 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])

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPS: __m128 _mm_sqrt_ps(__m128 a)

SQRTPS: __m256 _mm256_sqrt_ps (__m256 a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.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*	RM	V/V	SSE2	Computes square root of the low double-precision floating-point value in *xmm2/m64* and stores the results in *xmm1*.
VEX.NDS.LIG.F2.0F.WIG 51/r VSQRTSD *xmm1,xmm2, xmm3/m64*	RVM	V/V	AVX	Computes square root of the low double-precision floating point value in *xmm3/m64* and stores the results in *xmm2*. Also, upper double precision floating-point value (bits[127:64]) from *xmm2* are copied to *xmm1*[127:64].

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 the square root of the low double-precision floating-point value in the source operand (second operand) and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a scalar double-precision floating-point operation.

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

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

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

Operation

SQRTSD (128-bit Legacy SSE version)

```
DEST[63:0] ← SQRT(SRC[63:0])
DEST[VLMAX-1:64] (Unmodified)
```

VSQRTSD (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD: __m128d _mm_sqrt_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /*r* SQRTSS *xmm1*, *xmm2/m32*	RM	V/V	SSE	Computes square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*.
VEX.NDS.LIG.F3.0F.WIG 51/r VSQRTSS *xmm1, xmm2, xmm3/m32*	RVM	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].

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 the square root of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order double-words of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a scalar single-precision floating-point operation.

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

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

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

Operation

SQRTSS (128-bit Legacy SSE version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[VLMAX-1:32] (Unmodified)

VSQRTSS (VEX.128 encoded version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSS: __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

STAC—Set AC Flag in EFLAGS Register

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F 01 CB	STAC	NP	Valid	Valid	Set the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	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	NP	Valid	Valid	Set CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	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	NP	Valid	Valid	Set DF flag.

Instruction Operand Encoding

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

Description

If protected-mode virtual interrupts are not enabled, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized[1]. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts (and SMIs) may be blocked for one macroinstruction following an STI.

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

Table 4-16 indicates the action of the STI instruction depending on the processor's mode of operation and the CPL/IOPL settings of the running program or procedure.

Operation is the same in all modes.

Table 4-16. Decision Table for STI Results

CR0.PE	EFLAGS.VM	EFLAGS.IOPL	CS.CPL	CR4.PVI	EFLAGS.VIP	CR4.VME	STI Result
0	X	X	X	X	X	X	**IF = 1**
1	0	≥ CPL	X	X	X	X	**IF = 1**
1	0	< CPL	3	1	X	X	**VIF = 1**
1	0	< CPL	< 3	X	X	X	**GP Fault**
1	0	< CPL	X	0	X	X	**GP Fault**
1	0	< CPL	X	X	1	X	**GP Fault**
1	1	3	X	X	X	X	**IF = 1**
1	1	< 3	X	X	0	1	**VIF = 1**
1	1	< 3	X	X	1	X	**GP Fault**
1	1	< 3	X	X	X	0	**GP Fault**

NOTES:

X = This setting has no impact.

1. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.

   In the following instruction sequence, interrupts may be recognized before RET executes:

   STI
   STI
   RET

Operation

```
IF PE = 0  (* Executing in real-address mode *)
    THEN
        IF ← 1; (* Set Interrupt Flag *)
    ELSE  (* Executing in protected mode or virtual-8086 mode *)
        IF VM = 0  (* Executing in protected mode*)
            THEN
                IF IOPL ≥ CPL
                    THEN
                        IF ← 1;  (* Set Interrupt Flag *)
                ELSE
                    IF (IOPL < CPL) and (CPL = 3) and (PVI = 1)
                        THEN
                            VIF ← 1;  (* Set Virtual Interrupt Flag *)
                        ELSE
                            #GP(0);
                    FI;
                FI;
            ELSE  (* Executing in Virtual-8086 mode *)
                IF IOPL = 3
                    THEN
                        IF ← 1;  (* Set Interrupt Flag *)
                ELSE
                    IF ((IOPL < 3) and (VIP = 0) and (VME = 1))
                        THEN
                            VIF ← 1;  (* Set Virtual Interrupt Flag *)
                    ELSE
                        #GP(0); (* Trap to virtual-8086 monitor *)
                    FI;)
                FI;
        FI;
FI;
```

Flags Affected

The IF flag is set to 1; or the VIF flag is set to 1. Other flags are unaffected.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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.

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.
#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.
#SS(U)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C *ib*	SUB AL, i*mm8*	I	Valid	Valid	Subtract *imm8* from AL.
2D *iw*	SUB AX, i*mm16*	I	Valid	Valid	Subtract *imm16* from AX.
2D *id*	SUB EAX, i*mm32*	I	Valid	Valid	Subtract *imm32* from EAX.
REX.W + 2D *id*	SUB RAX, i*mm32*	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	imm8/26/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/26/32	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*	RM	V/V	SSE2	Subtract packed double-precision floating-point values in *xmm2/m128* from *xmm1*.
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed double-precision floating-point values in *xmm3/mem* from *xmm2* and stores result in *xmm1*.
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Subtract packed double-precision floating-point values in *ymm3/mem* from *ymm2* and stores result in *ymm1*.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the two packed double-precision floating-point values in the source operand (second operand) from the two packed double-precision floating-point values in the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

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

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

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

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

Operation

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

VSUBPD (VEX.128 encoded version)

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

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]

Intel C/C++ Compiler Intrinsic Equivalent

SUBPD: __m128d _mm_sub_pd (m128d a, m128d b)

VSUBPD: __m256d _mm256_sub_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 5C /*r* SUBPS *xmm1 xmm2/m128*	RM	V/V	SSE	Subtract packed single-precision floating-point values in *xmm2/mem* from *xmm1*.
VEX.NDS.128.0F.WIG 5C /r VSUBPS *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract packed single-precision floating-point values in *xmm3/mem* from *xmm2* and stores result in *xmm1*.
VEX.NDS.256.0F.WIG 5C /r VSUBPS *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Subtract packed single-precision floating-point values in *ymm3/mem* from *ymm2* and stores result in *ymm1*.

Instruction Operand Encoding

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

Description

Performs a SIMD subtract of the four packed single-precision floating-point values in the source operand (second operand) from the four packed single-precision floating-point values in the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

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

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

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

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

Operation

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

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[VLMAX-1:128] ← 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].

Intel C/C++ Compiler Intrinsic Equivalent

SUBPS: __m128 _mm_sub_ps(__m128 a, __m128 b)

VSUBPS: __m256 _mm256_sub_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

SUBSD—Subtract Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /*r* SUBSD *xmm1, xmm2/m64*	RM	V/V	SSE2	Subtracts the low double-precision floating-point values in *xmm2/mem64* from *xmm1*.
VEX.NDS.LIG.F2.0F.WIG 5C /r VSUBSD *xmm1,xmm2, xmm3/m64*	RVM	V/V	AVX	Subtract the low double-precision floating-point value in *xmm3/mem* from *xmm2* and store the result in *xmm1*.

Instruction Operand Encoding

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

Description

Subtracts the low double-precision floating-point value in the source operand (second operand) from the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a scalar double-precision floating-point operation.

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

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

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

Operation

SUBSD (128-bit Legacy SSE version)

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

VSUBSD (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

SUBSD: __m128d _mm_sub_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /*r* SUBSS *xmm1, xmm2/m32*	RM	V/V	SSE	Subtract the lower single-precision floating-point values in *xmm2/m32* from *xmm1*.
VEX.NDS.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/mem from xmm2 and store the result in xmm1.

Instruction Operand Encoding

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

Description

Subtracts the low single-precision floating-point value in the source operand (second operand) from the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for an illustration of a scalar single-precision floating-point operation.

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

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

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

Operation

SUBSS (128-bit Legacy SSE version)

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

VSUBSS (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

SUBSS: __m128 _mm_sub_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 01 F8	SWAPGS	NP	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
NP	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.
	If the LOCK prefix is used.

SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	NP	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

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

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	NP	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

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

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	NP	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	NP	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
NP	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.

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 ≠ 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 ≠ 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	NP	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + 0F 07	SYSRET	NP	Valid	Invalid	Return to 64-bit mode from fast system call

Instruction Operand Encoding

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

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) OR (RCX is not canonical) THEN #GP(0); FI;
```

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.

```
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;          (* Clear RF, VM, reserved bits; set bit 2 *)

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 ← 0;

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 RCX contains a non-canonical address.

TEST—Logical Compare

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A8 *ib*	TEST AL, *imm8*	I	Valid	Valid	AND *imm8* with AL; set SF, ZF, PF according to result.
A9 *iw*	TEST AX, *imm16*	I	Valid	Valid	AND *imm16* with AX; set SF, ZF, PF according to result.
A9 *id*	TEST EAX, *imm32*	I	Valid	Valid	AND *imm32* with EAX; set SF, ZF, PF according to result.
REX.W + A9 *id*	TEST RAX, *imm32*	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.

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*	RM	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*	RM	V/V	BMI1	Count the number of trailing zero bits in *r/m32*, return result in *r32*.
REX.W + F3 0F BC /r TZCNT *r64, r/m64*	RM	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*	RM	V/V	SSE2	Compares (unordered) the low double-precision floating-point values in *xmm1* and *xmm2/m64* and set the EFLAGS accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD *xmm1, xmm2/m64*	RM	V/V	AVX	Compare low double precision floating-point values in *xmm1* and *xmm2/mem64* and set the EFLAGS flags accordingly.

Instruction Operand Encoding

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

Description

Performs an unordered compare of the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

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

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

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

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

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

Operation

```
RESULT ← UnorderedCompare(SRC1[63:0] < > SRC2[63:0]) {
(* Set EFLAGS *)
CASE (RESULT) OF
    UNORDERED:          ZF, PF, CF ← 111;
    GREATER_THAN:       ZF, PF, CF ← 000;
    LESS_THAN:          ZF, PF, CF ← 001;
    EQUAL:              ZF, PF, CF ← 100;
ESAC;
OF, AF, SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_ucomieq_sd(__m128d a, __m128d b)
int _mm_ucomilt_sd(__m128d a, __m128d b)
int _mm_ucomile_sd(__m128d a, __m128d b)
int _mm_ucomigt_sd(__m128d a, __m128d b)
int _mm_ucomige_sd(__m128d a, __m128d b)
int _mm_ucomineq_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv ≠ 1111B.

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
0F 2E /r UCOMISS *xmm1, xmm2/m32*	RM	V/V	SSE	Compare lower single-precision floating-point value in *xmm1* register with lower single-precision floating-point value in *xmm2/mem* and set the status flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS *xmm1, xmm2/m32*	RM	V/V	AVX	Compare low single precision floating-point values in *xmm1* and *xmm2/mem32* and set the EFLAGS flags accordingly.

Instruction Operand Encoding

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

Description

Performs an unordered compare of the single-precision floating-point values in the low doublewords of the source operand 1 (first operand) and the source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

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

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

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

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

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

Operation

```
RESULT ← UnorderedCompare(SRC1[31:0] <> SRC2[31:0]) {
(* Set EFLAGS *)
CASE (RESULT) OF
    UNORDERED:          ZF,PF,CF ← 111;
    GREATER_THAN:       ZF,PF,CF ← 000;
    LESS_THAN:          ZF,PF,CF ← 001;
    EQUAL:              ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_ucomieq_ss(__m128 a, __m128 b)
int _mm_ucomilt_ss(__m128 a, __m128 b)
int _mm_ucomile_ss(__m128 a, __m128 b)
int _mm_ucomigt_ss(__m128 a, __m128 b)
int _mm_ucomige_ss(__m128 a, __m128 b)
```

int _mm_ucomineq_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv ≠ 1111B.

UD2—Undefined Instruction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 0B	UD2	NP	Valid	Valid	Raise invalid opcode exception.

Instruction Operand Encoding

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

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcode for this instruction is 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 UD2 instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD2 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.

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*	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of *xmm1* and *xmm2/m128*.
VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of *xmm2* and *xmm3/m128*.
VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD *ymm1,ymm2, ymm3/m256*	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values from high quadwords of *ymm2* and *ymm3/m256*.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the high double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-23.

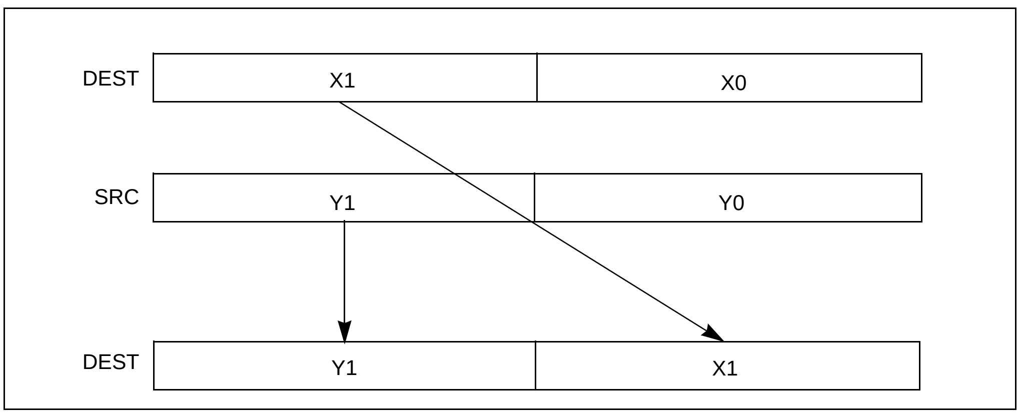

Figure 4-23. UNPCKHPD Instruction High Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

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

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

Operation

UNPCKHPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[VLMAX-1:128] (Unmodified)

VUNPCKHPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[VLMAX-1:128] ← 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]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD: __m128d _mm_unpackhi_pd(__m128d a, __m128d b)

UNPCKHPD: __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 15 /*r* UNPCKHPS *xmm1, xmm2/m128*	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of *xmm1* and *xmm2/mem* into *xmm1*.
VEX.NDS.128.0F.WIG 15 /r VUNPCKHPS *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of *xmm2* and *xmm3/m128*.
VEX.NDS.256.0F.WIG 15 /r VUNPCKHPS *ymm1,ymm2,ymm3/m256*	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of *ymm2* and *ymm3/m256*.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the high-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-24. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

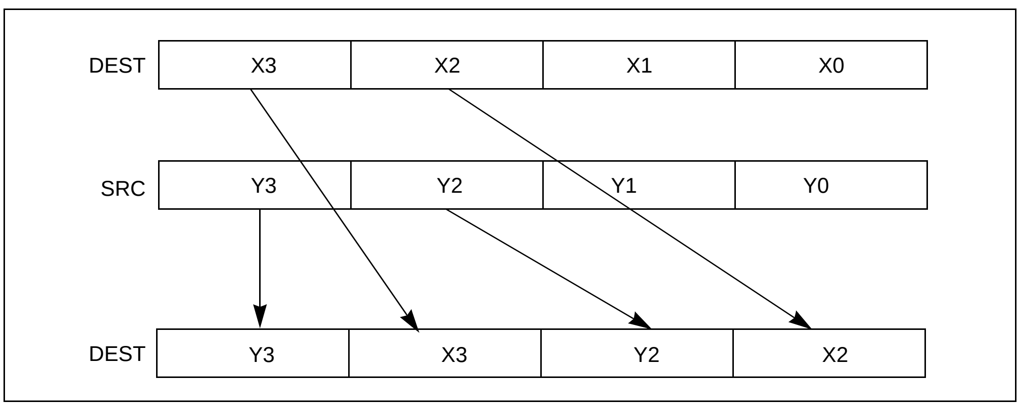

Figure 4-24. UNPCKHPS Instruction High Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

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

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 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]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS: __m128 _mm_unpackhi_ps(__m128 a, __m128 b)

UNPCKHPS: __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

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*	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of *xmm1* and *xmm2/m128*.
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of *xmm2* and *xmm3/m128*.
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD *ymm1,ymm2, ymm3/m256*	RVM	V/V	AVX	Unpacks and Interleaves double precision floating-point values low high quadwords of *ymm2* and *ymm3/m256*.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-25. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

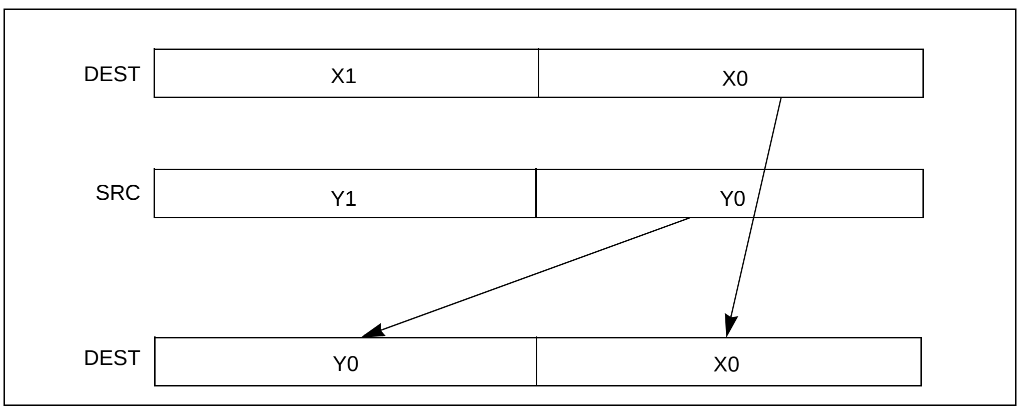

Figure 4-25. UNPCKLPD Instruction Low Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

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

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

Operation

UNPCKLPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] (Unmodified)

VUNPCKLPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] ← 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]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD: __m128d _mm_unpacklo_pd(__m128d a, __m128d b)

UNPCKLPD: __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 14 /r UNPCKLPS *xmm1, xmm2/m128*	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of *xmm1* and *xmm2/mem* into *xmm1*.
VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of *xmm2* and *xmm3/m128*.
VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS *ymm1,ymm2,ymm3/m256*	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of *ymm2* and *ymm3/m256*.

Instruction Operand Encoding

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

Description

Performs an interleaved unpack of the low-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-26. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

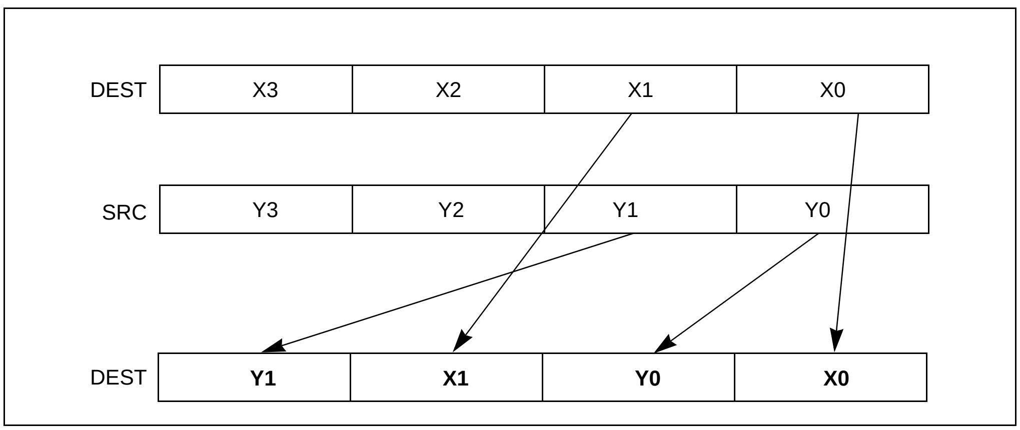

Figure 4-26. UNPCKLPS Instruction Low Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

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

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

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VUNPCKLPS (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]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS: __m128 _mm_unpacklo_ps(__m128 a, __m128 b)

UNPCKLPS: __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VBROADCAST—Broadcast Floating-Point Data

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS *xmm1, m32*	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in *xmm1.*
VEX.256.66.0F38.W0 18 /r VBROADCASTSS *ymm1, m32*	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in *ymm1.*
VEX.256.66.0F38.W0 19 /r VBROADCASTSD *ymm1, m64*	RM	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in *ymm1.*
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 *ymm1, m128*	RM	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in *ymm1.*
VEX.128.66.0F38.W0 18/r VBROADCASTSS *xmm1, xmm2*	RM	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in *xmm1.*
VEX.256.66.0F38.W0 18 /r VBROADCASTSS *ymm1, xmm2*	RM	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in *ymm1.*
VEX.256.66.0F38.W0 19 /r VBROADCASTSD *ymm1, xmm2*	RM	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations 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

Load floating point values from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

VBROADCASTSD and VBROADCASTF128 are only supported as 256-bit wide versions. VBROADCASTSS is supported in both 128-bit and 256-bit wide versions.

Memory and register source operand syntax support of 256-bit instructions depend on the processor's enumeration of the following conditions with respect to CPUID.1:ECX.AVX[bit 28] and CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]:

- If CPUID.1:ECX.AVX = 1 and CPUID.(EAX=07H, ECX=0H):EBX.AVX2 = 0: the destination operand is a YMM register. The source operand support can be either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD.
- If CPUID.1:ECX.AVX = 1 and CPUID.(EAX=07H, ECX=0H):EBX.AVX2 = 1: the destination operand is a YMM register. The source operand support can be a register or memory location.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. An attempt to execute VBROADCASTSD or VBROADCASTF128 encoded with VEX.L= 0 will cause an #UD exception. Attempts to execute any VBROADCAST* instruction with VEX.W = 1 will cause #UD.

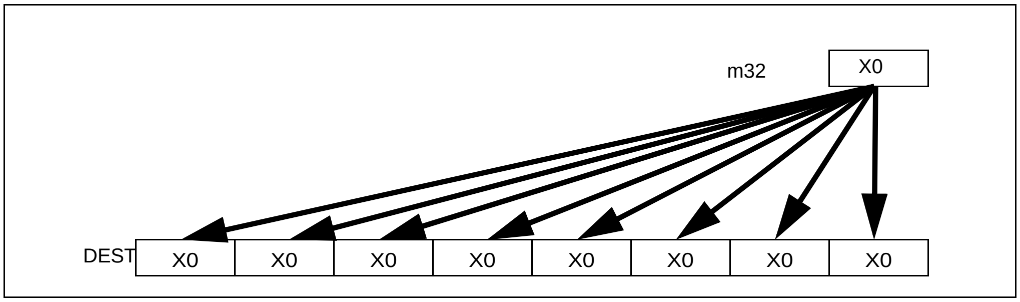

Figure 4-27. VBROADCASTSS Operation (VEX.256 encoded version)

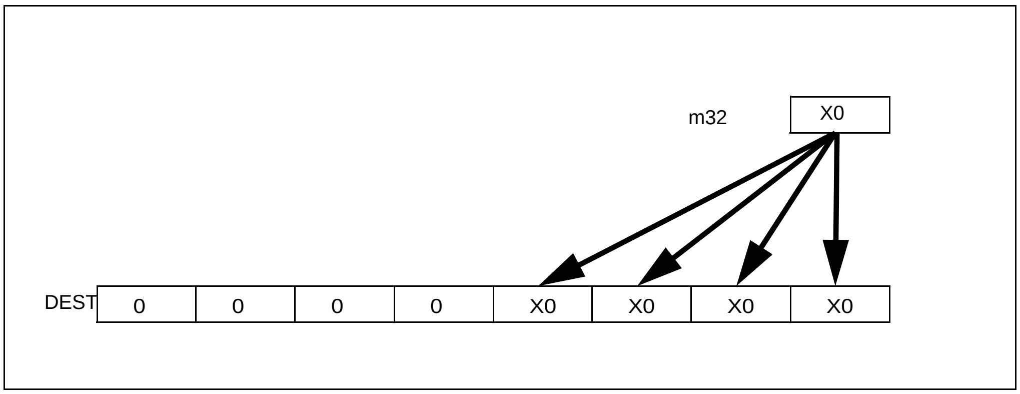

Figure 4-28. VBROADCASTSS Operation (128-bit version)

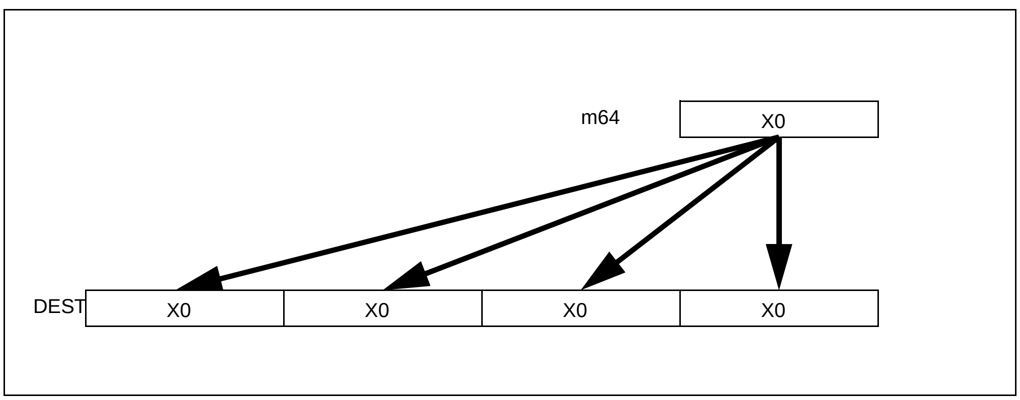

Figure 4-29. VBROADCASTSD Operation

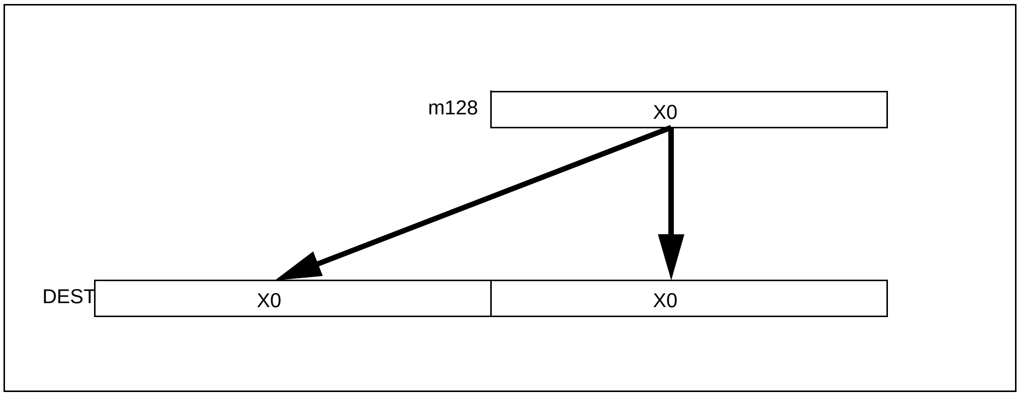

Figure 4-30. VBROADCASTF128 Operation

Operation

VBROADCASTSS (128 bit version)
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[VLMAX-1:128] ← 0

VBROADCASTSS (VEX.256 encoded version)
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp

VBROADCASTSD (VEX.256 encoded version)
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp

VBROADCASTF128
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[VLMAX-1:128] ← temp

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTSS: __m128 _mm_broadcast_ss(float *a);

VBROADCASTSS: __m256 _mm256_broadcast_ss(float *a);

VBROADCASTSD: __m256d _mm256_broadcast_sd(double *a);

VBROADCASTF128: __m256 _mm256_broadcast_ps(__m128 * a);

VBROADCASTF128: __m256d _mm256_broadcast_pd(__m128d * a);

Flags Affected

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0 for VBROADCASTSD,
	If VEX.L = 0 for VBROADCASTF128,
	If VEX.W = 1.

VCVTPH2PS—Convert 16-bit FP Values to Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 13 /r VCVTPH2PS *ymm1, xmm2/m128*	RM	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in *xmm2/m128* to packed single-precision floating-point value in *ymm1*.
VEX.128.66.0F38.W0 13 /r VCVTPH2PS *xmm1, xmm2/m64*	RM	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in *xmm2/m64* to packed single-precision floating-point value in *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

Converts four/eight packed half precision (16-bits) floating-point values in the low-order 64/128 bits of an XMM/YMM register or 64/128-bit memory location to four/eight packed single-precision floating-point values and writes the converted values into the destination XMM/YMM register.

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

128-bit version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (VLMAX-1:128) of the corresponding destination YMM register are zeroed.

256-bit version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv is reserved (must be 1111b).

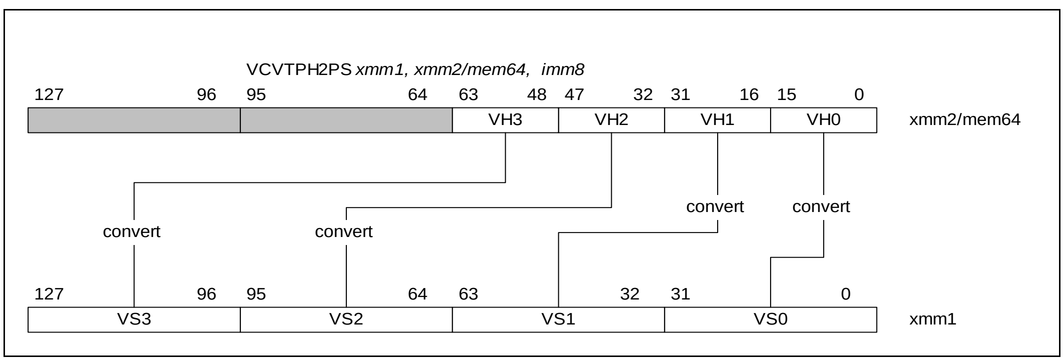

Figure 4-31. VCVTPH2PS (128-bit Version)

Operation

```
vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

VCVTPH2PS (VEX.256 encoded version)

DEST[31:0] ←vCvt_h2s(SRC1[15:0]);
DEST[63:32] ←vCvt_h2s(SRC1[31:16]);
DEST[95:64] ←vCvt_h2s(SRC1[47:32]);
DEST[127:96] ←vCvt_h2s(SRC1[63:48]);
DEST[159:128] ←vCvt_h2s(SRC1[79:64]);
DEST[191:160] ←vCvt_h2s(SRC1[95:80]);
DEST[223:192] ←vCvt_h2s(SRC1[111:96]);
DEST[255:224] ←vCvt_h2s(SRC1[127:112]);

VCVTPH2PS (VEX.128 encoded version)

DEST[31:0] ←vCvt_h2s(SRC1[15:0]);
DEST[63:32] ←vCvt_h2s(SRC1[31:16]);
DEST[95:64] ←vCvt_h2s(SRC1[47:32]);
DEST[127:96] ←vCvt_h2s(SRC1[63:48]);
DEST[VLMAX-1:128] ←0

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_cvtph_ps (__m128i m1);

__m256 _mm256_cvtph_ps (__m128i m1)

SIMD Floating-Point Exceptions

Invalid

Other Exceptions

Exceptions Type 11 (do not report #AC); additionally

#UD	If VEX.W=1.

VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH *xmm1/m128, ymm2, imm8*	MR	V/V	F16C	Convert eight packed single-precision floating-point value in *ymm2* to packed half-precision (16-bit) floating-point value in *xmm1/mem*. *Imm8* provides rounding controls.
VEX.128.66.0F3A.W0.1D /r ib VCVTPS2PH *xmm1/m64, xmm2, imm8*	MR	V/V	F16C	Convert four packed single-precision floating-point value in *xmm2* to packed half-precision (16-bit) floating-point value in *xmm1/mem*. *Imm8* provides rounding controls.

Instruction Operand Encoding

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

Description

Convert four or eight packed single-precision floating values in first source operand to four or eight packed half-precision (16-bit) floating-point values. The rounding mode is specified using the immediate field (imm8).

Non-zero tiny results are converted to zero, denormals, or the smallest normalized half-precision floating-point value. MXCSR.FTZ is ignored. If a source element is denormal relative to input format with MXCSR.DAZ not set, DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

128-bit version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. The upper-bits vector register zeroing behavior of VEX prefix encoding still applies if the destination operand is a xmm register. So the upper bits (255:64) of corresponding YMM register are zeroed.

256-bit version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. The upper-bits vector register zeroing behavior of VEX prefix encoding still applies if the destination operand is a xmm register. So the upper bits (255:128) of the corresponding YMM register are zeroed.

Note: VEX.vvvv is reserved (must be 1111b).

The diagram below illustrates how data is converted from four packed single precision (in 128 bits) to four half precision (in 64 bits) FP values.

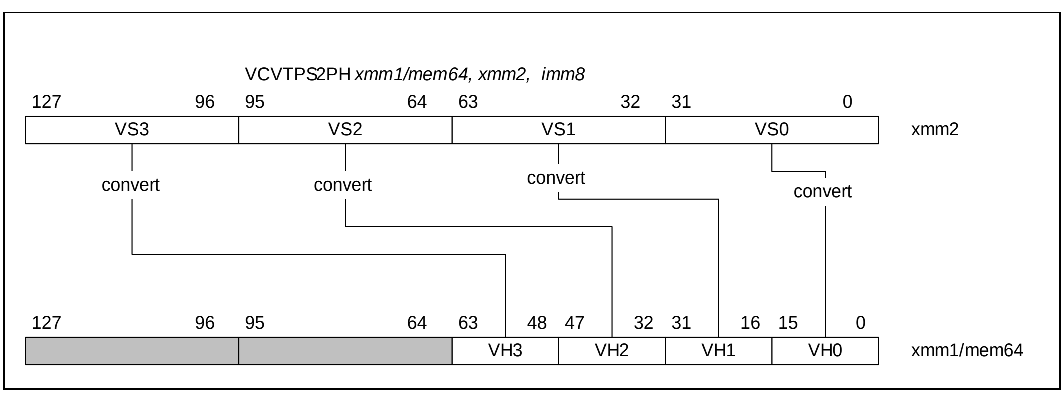

Figure 4-32. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that controls rounding operation. The effect and encoding of RC field are listed in Table 4-17.

Table 4-17. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN // using Imm[1:0] for rounding control, see Table 4-17
        RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE // using MXCSR.RC for rounding control
        RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

VCVTPS2PH (VEX.256 encoded version)

```
DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[79:64] ← vCvt_s2h(SRC1[159:128]);
DEST[95:80] ← vCvt_s2h(SRC1[191:160]);
DEST[111:96] ← vCvt_s2h(SRC1[223:192]);
DEST[127:112] ← vCvt_s2h(SRC1[255:224]);
DEST[255:128] ← 0; // if DEST is a register
```

VCVTPS2PH (VEX.128 encoded version)

```
DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[VLMAX-1:64] ←0; // if DEST is a register
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

__m128i _mm_cvtps_ph (__m128 m1, const int imm);

__m128i _mm256_cvtps_ph(__m256 m1, const int imm);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

Other Exceptions

Exceptions Type 11 (do not report #AC); additionally

#UD	If VEX.W=1.

VERR/VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /4	VERR *r/m16*	M	Valid	Valid	Set ZF=1 if segment specified with *r/m16* can be read.
0F 00 /5	VERW *r/m16*	M	Valid	Valid	Set ZF=1 if segment specified with *r/m16* can be written.

Instruction Operand Encoding

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

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment’s DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

Operation

```
IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF ← 0; FI;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
    THEN
        ZF ← 0;
    ELSE
        IF ((Instruction = VERR) and (Segment readable))
        or ((Instruction = VERW) and (Segment writable))
            THEN
                ZF ← 1;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

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

Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode.
	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a 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.

VEXTRACTF128 — Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 *xmm1/m128, ymm2, imm8*	MR	V/V	AVX	Extract 128 bits of packed floating-point values from *ymm2* and store results in *xmm1/mem.*

Instruction Operand Encoding

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

Description

Extracts 128-bits of packed floating-point values from the source operand (second operand) at an 128-bit offset from imm8[0] into the destination operand (first operand). The destination may be either an XMM register or an 128-bit memory location.

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

The high 7 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTF128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
```

VEXTRACTF128 (register destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXTRACTF128:    __m128 _mm256_extractf128_ps (__m256 a, int offset);
VEXTRACTF128:    __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128:    __m128i_mm256_extractf128_si256(__m256i a, int offset);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L= 0
	If VEX.W=1.

VEXTRACTI128 — Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 *xmm1/m128, ymm2, imm8*	RMI	V/V	AVX2	Extract 128 bits of integer data from *ymm2* and store results in *xmm1/mem*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

Extracts 128-bits of packed integer values from the source operand (second operand) at a 128-bit offset from imm8[0] into the destination operand (first operand). The destination may be either an XMM register or a 128-bit memory location.

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

The high 7 bits of the immediate are ignored.

An attempt to execute VEXTRACTI128 encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTI128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
```

VEXTRACTI128 (register destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTI128: __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	IF VEX.L = 0, If VEX.W = 1.

VFMADD132PD/VFMADD213PD/VFMADD231PD — Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 98 /r VFMADD132PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm2/mem*, add to *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 A8 /r VFMADD213PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm1*, add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 B8 /r VFMADD231PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm1* and *xmm2/mem*, add to *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W1 98 /r VFMADD132PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm2/mem*, add to *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 A8 /r VFMADD213PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm1*, add to *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 B8 /r VFMADD231PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm1* and *ymm2/mem*, add to *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD213PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD231PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD132PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD213PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD231PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132PS/VFMADD213PS/VFMADD231PS — Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 98 /r VFMADD132PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm2/mem*, add to *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 A8 /r VFMADD213PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm1*, add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 B8 /r VFMADD231PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm1* and *xmm2/mem*, add to *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W0 98 /r VFMADD132PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm2/mem*, add to *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 A8 /r VFMADD213PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm1*, add to *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 B8 /r VFMADD231PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm1* and *ymm2/mem*, add to *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PS: __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD213PS: __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD231PS: __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD132PS: __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

VFMADD213PS: __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

VFMADD231PS: __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132SD/VFMADD213SD/VFMADD231SD — Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 99 /r VFMADD132SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm2/mem*, add to *xmm1* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W1 A9 /r VFMADD213SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm1*, add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W1 B9 /r VFMADD231SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm1* and *xmm2/mem*, add to *xmm0* and put result in *xmm0*.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-add computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, *Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFMADD213SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFMADD231SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SD: __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

VFMADD213SD: __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

VFMADD231SD: __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADD132SS/VFMADD213SS/VFMADD231SS — Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 99 /r VFMADD132SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm0* and *xmm2/mem*, add to *xmm1* and put result in *xmm0.*
VEX.DDS.LIG.128.66.0F38.W0 A9 /r VFMADD213SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm0* and *xmm1*, add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W0 B9 /r VFMADD231SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm1* and *xmm2/mem*, add to *xmm0* and put result in *xmm0*.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMADD213SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMADD231SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[63:0] + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SS: __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

VFMADD213SS: __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

VFMADD231SS: __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD — Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm2/mem*, add/subtract elements in *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm1*, add/subtract elements in *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm1* and *xmm2/mem*, add/subtract elements in *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm2/mem*, add/subtract elements in *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm1*, add/subtract elements in *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm1* and *ymm2/mem*, add/subtract elements in *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFMADDSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192]
FI
```

VFMADDSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192]
FI
```

VFMADDSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB213PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB231PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB132PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB213PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB231PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS — Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm2/mem*, add/subtract elements in *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm1*, add/subtract elements in *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm1* and *xmm2/mem*, add/subtract elements in *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm2/mem*, add/subtract elements in *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm1*, add/subtract elements in *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm1* and *ymm2/mem*, add/subtract elements in *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFMADDSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, *Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PS: __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB213PS: __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB231PS: __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB132PS: __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

VFMADDSUB213PS: __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

VFMADDSUB231PS: __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD — Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm2/mem*, subtract/add elements in *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm1*, subtract/add elements in *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm1* and *xmm2/mem*, subtract/add elements in *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm2/mem*, subtract/add elements in *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm1*, subtract/add elements in *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm1* and *ymm2/mem*, subtract/add elements in *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFMSUBADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192]
FI
```

VFMSUBADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192]
FI
```

VFMSUBADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD213PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD231PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD132PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD213PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD231PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS — Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm2/mem*, subtract/add elements in *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm1*, subtract/add elements in *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm1* and *xmm2/mem*, subtract/add elements in *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm2/mem*, subtract/add elements in *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm1*, subtract/add elements in *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm1* and *ymm2/mem*, subtract/add elements in *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFMSUBADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, *Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PS: __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD213PS: __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD231PS: __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD132PS: __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

VFMSUBADD213PS: __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

VFMSUBADD231PS: __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD — Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9A /r VFMSUB132PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm2/mem*, subtract *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 AA /r VFMSUB213PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm1*, subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 BA /r VFMSUB231PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm1* and *xmm2/mem*, subtract *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W1 9A /r VFMSUB132PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm2/mem*, subtract *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 AA /r VFMSUB213PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm1*, subtract *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 BA /r VFMSUB231PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm1* and *ymm2/mem*, subtract *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PD: __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB213PD: __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB231PD: __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB132PD: __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

VFMSUB213PD: __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

VFMSUB231PD: __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS — Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9A /r VFMSUB132PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm2/mem*, subtract *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 AA /r VFMSUB213PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm1*, subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 BA /r VFMSUB231PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm1* and *xmm2/mem*, subtract *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W0 9A /r VFMSUB132PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm2/mem*, subtract *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 AA /r VFMSUB213PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm1*, subtract *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.0 BA /r VFMSUB231PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm1* and *ymm2/mem*, subtract *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PS: __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB213PS: __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB231PS: __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB132PS: __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

VFMSUB213PS: __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

VFMSUB231PS: __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD — Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9B /r VFMSUB132SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm2/mem*, subtract *xmm1* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W1 AB /r VFMSUB213SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm1*, subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W1 BB /r VFMSUB231SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm1* and *xmm2/mem*, subtract *xmm0* and put result in *xmm0*.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFMSUB213SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFMSUB231SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SD: __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

VFMSUB213SD: __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

VFMSUB231SD: __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS — Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9B /r VFMSUB132SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm0* and *xmm2/mem*, subtract *xmm1* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W0 AB /r VFMSUB213SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm0* and *xmm1*, subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W0 BB /r VFMSUB231SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm1* and *xmm2/mem*, subtract *xmm0* and put result in *xmm0*.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMSUB213SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMSUB231SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SS: __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

VFMSUB213SS: __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

VFMSUB231SS: __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD — Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9C /r VFNMADD132PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm2/mem*, negate the multiplication result and add to *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 AC /r VFNMADD213PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm1*, negate the multiplication result and add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 BC /r VFNMADD231PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm1* and *xmm2/mem*, negate the multiplication result and add to *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W1 9C /r VFNMADD132PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm2/mem*, negate the multiplication result and add to *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 AC /r VFNMADD213PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm1*, negate the multiplication result and add to *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 BC /r VFNMADD231PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm1* and *ymm2/mem*, negate the multiplication result and add to *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFNMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
```

```
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PD: __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD213PD: __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD231PD: __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD132PD: __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

VFNMADD213PD: __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

VFNMADD231PD: __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS — Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9C /r VFNMADD132PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm2/mem*, negate the multiplication result and add to *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 AC /r VFNMADD213PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm1*, negate the multiplication result and add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 BC /r VFNMADD231PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm1* and *xmm2/mem*, negate the multiplication result and add to *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W0 9C /r VFNMADD132PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm2/mem*, negate the multiplication result and add to *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 AC /r VFNMADD213PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm1*, negate the multiplication result and add to *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.0 BC /r VFNMADD231PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm1* and *ymm2/mem*, negate the multiplication result and add to *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFNMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
```

FI

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD213PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD231PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD132PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD213PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD231PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD — Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9D /r VFNMADD132SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm2/mem*, negate the multiplication result and add to *xmm1* and put result in *xmm0.*
VEX.DDS.LIG.128.66.0F38.W1 AD /r VFNMADD213SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm1*, negate the multiplication result and add to *xmm2/mem* and put result in *xmm0.*
VEX.DDS.LIG.128.66.0F38.W1 BD /r VFNMADD231SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm1* and *xmm2/mem*, negate the multiplication result and add to *xmm0* and put result in *xmm0.*

Instruction Operand Encoding

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

Description

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFNMADD213SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFNMADD231SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132SD: __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

VFNMADD213SD: __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

VFNMADD231SD: __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS — Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9D /r VFNMADD132SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm0* and *xmm2/mem*, negate the multiplication result and add to *xmm1* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W0 AD /r VFNMADD213SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm0* and *xmm1*, negate the multiplication result and add to *xmm2/mem* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W0 BD /r VFNMADD231SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from *xmm1* and *xmm2/mem*, negate the multiplication result and add to *xmm0* and put result in *xmm0*.

Instruction Operand Encoding

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

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFNMADD213SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFNMADD231SS DEST, SRC2, SRC3
DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132SS: __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

VFNMADD213SS: __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

VFNMADD231SS: __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD — Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9E /r VFNMSUB132PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm2/mem*, negate the multiplication result and subtract *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 AE /r VFNMSUB213PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm0* and *xmm1*, negate the multiplication result and subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W1 BE /r VFNMSUB231PD *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed double-precision floating-point values from *xmm1* and *xmm2/mem*, negate the multiplication result and subtract *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W1 9E /r VFNMSUB132PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm2/mem*, negate the multiplication result and subtract *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 AE /r VFNMSUB213PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm0* and *ymm1*, negate the multiplication result and subtract *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W1 BE /r VFNMSUB231PD *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed double-precision floating-point values from *ymm1* and *ymm2/mem*, negate the multiplication result and subtract *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFNMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, *Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
```

```
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB213PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB231PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB132PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB213PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB231PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS — Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9E /r VFNMSUB132PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm2/mem*, negate the multiplication result and subtract *xmm1* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 AE /r VFNMSUB213PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm0* and *xmm1*, negate the multiplication result and subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.128.66.0F38.W0 BE /r VFNMSUB231PS *xmm0, xmm1, xmm2/m128*	A	V/V	FMA	Multiply packed single-precision floating-point values from *xmm1* and *xmm2/mem*, negate the multiplication result and subtract *xmm0* and put result in *xmm0*.
VEX.DDS.256.66.0F38.W0 9E /r VFNMSUB132PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm2/mem*, negate the multiplication result and subtract *ymm1* and put result in *ymm0*.
VEX.DDS.256.66.0F38.W0 AE /r VFNMSUB213PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm0* and *ymm1*, negate the multiplication result and subtract *ymm2/mem* and put result in *ymm0*.
VEX.DDS.256.66.0F38.0 BE /r VFNMSUB231PS *ymm0, ymm1, ymm2/m256*	A	V/V	FMA	Multiply packed single-precision floating-point values from *ymm1* and *ymm2/mem,* negate the multiplication result and subtract *ymm0* and put result in *ymm0*.

Instruction Operand Encoding

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

Description

VFNMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a

XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, "FMA Instruction Operand Order and Arithmetic Behavior" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
```

```
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB213PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB231PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB132PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

VFNMSUB213PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

VFNMSUB231PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD — Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9F /r VFNMSUB132SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm2/mem*, negate the multiplication result and subtract *xmm1* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W1 AF /r VFNMSUB213SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm0* and *xmm1*, negate the multiplication result and subtract *xmm2/mem* and put result in *xmm0*.
VEX.DDS.LIG.128.66.0F38.W1 BF /r VFNMSUB231SD *xmm0, xmm1, xmm2/m64*	A	V/V	FMA	Multiply scalar double-precision floating-point value from *xmm1* and *xmm2/mem*, negate the multiplication result and subtract *xmm0* and put result in *xmm0*.

Instruction Operand Encoding

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

Description

VFNMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFNMSUB213SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

VFNMSUB231SD DEST, SRC2, SRC3
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SD: __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

VFNMSUB213SD: __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

VFNMSUB231SD: __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS — Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9F /r VFNMSUB132SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 AF /r VFNMSUB213SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 BF /r VFNMSUB231SS *xmm0, xmm1, xmm2/m32*	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([VLMAX-1:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 14.5.1, “FMA Instruction Operand Order and Arithmetic Behavior” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
```

DEST[VLMAX-1:128] ← 0

VFNMSUB213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFNMSUB231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SS: __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

VFNMSUB213SS: __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

VFNMSUB231SS: __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather double-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather double-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD *ymm1, vm32x, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather double-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD *ymm1, vm64y, ymm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather double-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VGATHERDPD (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
```

```
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERQPD (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERQPD (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERDPD (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
```

```
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);

VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);

VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);

VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS *ymm1, vm32y, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32y*, gather single-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS *xmm1, vm64y, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VGATHERDPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
```

```
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERQPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERDPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERQPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
```

```
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
         DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);

VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);

VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);

VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);

VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERQPS: __m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);

VGATHERQPS: __m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD *ymm1, vm32y, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32y*, gather dword from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD *xmm1, vm64y, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VPGATHERDD (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
```

VPGATHERQD (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
         MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
         MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
         DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERDD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
         MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
         MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
         DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)
```

VPGATHERQD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
         MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
         MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
```

```
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDD: __m256i _mm256_i32gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);

VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather qword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather qword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ *ymm1, vm32x, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather qword values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ *ymm1, vm64y, ymm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather qword values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VPGATHERDQ (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    i ←j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERQQ (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERDQ (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 32;
    i ← j * 64;
```

```
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDQ: __m128i _mm_i32gather_epi64 (int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDQ: __m256i _mm256_i32gather_epi64 (int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, int64 const * base, __m128i index, __m256i mask, const int scale);

VPGATHERQQ: __m128i _mm_i64gather_epi64 (int64 const * base, __m128i index, const int scale);

VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQQ: __m256i _mm256_i64gather_epi64 (int64 const * base, __m256i index, const int scale);

VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, int64 const * base, __m256i index, __m256i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VINSERTF128 — Insert Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 *ymm1, ymm2, xmm3/m128, imm8*	RVM	V/V	AVX	Insert a single precision floating-point value selected by *imm8* from *xmm3/m128* into ymm2 at the specified destination element specified by *imm8* and zero out destination elements in *ymm1* as indicated in *imm8*.

Instruction Operand Encoding

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

Description

Performs an insertion of 128-bits of packed floating-point values from the second source operand (third operand) into an the destination operand (first operand) at an 128-bit offset from imm8[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

Operation

```
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ← SRC2[127:0]
    1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ←TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTF128: __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);

VINSERTF128: __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);

VINSERTF128: __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.W = 1.

VINSERTI128 — Insert Packed Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 *ymm1, ymm2, xmm3/m128, imm8*	RVMI	V/V	AVX2	Insert 128-bits of integer data from *xmm3/mem* and the remaining values from *ymm2* into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Performs an insertion of 128-bits of packed integer data from the second source operand (third operand) into an the destination operand (first operand) at a 128-bit offset from imm8[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

VEX.L must be 1; an attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI128

```
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ←SRC2[127:0]
    1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ←TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTI128: __m256i _mm256_inserti128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS *xmm1, xmm2, m128*	RVM	V/V	AVX	Conditionally load packed single-precision values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS *ymm1, ymm2, m256*	RVM	V/V	AVX	Conditionally load packed single-precision values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD *xmm1, xmm2, m128*	RVM	V/V	AVX	Conditionally load packed double-precision values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD *ymm1, ymm2, m256*	RVM	V/V	AVX	Conditionally load packed double-precision values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS *m128, xmm1, xmm2*	MVR	V/V	AVX	Conditionally store packed single-precision values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS *m256, ymm1, ymm2*	MVR	V/V	AVX	Conditionally store packed single-precision values from *ymm2* using mask in *ymm1*.
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD *m128, xmm1, xmm2*	MVR	V/V	AVX	Conditionally store packed double-precision values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD *m256, ymm1, ymm2*	MVR	V/V	AVX	Conditionally store packed double-precision values from *ymm2* using mask in *ymm1*.

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
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VMASKMOVPS - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VMASKMOVPD - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VMASKMOVPD - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

VMASKMOVPS - 256-bit store

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

VMASKMOVPD - 128-bit store

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]

VMASKMOVPD - 256-bit store

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

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)

void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)

__m256d _mm256_maskload_pd(double *a, __m256i mask);

void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);

__m128 _mm128_maskload_ps(float const *a, __m128i mask)

void _mm128_maskstore_ps(float *a, __m128i mask, __m128 b)

__m128d _mm128_maskload_pd(double *a, __m128i mask);

void _mm128_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations);
additionally

#UD	If VEX.W = 1.

VPBLENDD — Blend Packed Dwords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLENDD *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX2	Select dwords from *xmm2* and *xmm3/m128* from mask specified in *imm8* and store the values into *xmm1*.
VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLENDD *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Select dwords 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
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Dword elements 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 is 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 (VLMAX-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

VPBLENDD (VEX.256 encoded version)

```
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] ← SRC2[159:128]
ELSE DEST[159:128] ← SRC1[159:128]
IF (imm8[5] == 1) THEN DEST[191:160] ← SRC2[191:160]
ELSE DEST[191:160] ← SRC1[191:160]
IF (imm8[6] == 1) THEN DEST[223:192] ← SRC2[223:192]
ELSE DEST[223:192] ← SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] ← SRC2[255:224]
ELSE DEST[255:224] ← SRC1[255:224]
```

VPBLENDD (VEX.128 encoded version)

```
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDD: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)

VPBLENDD: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

VPBROADCAST—Broadcast Integer Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB *xmm1, xmm2/m8*	RM	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in *xmm1*.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB *ymm1, xmm2/m8*	RM	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in *ymm1*.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW *xmm1, xmm2/m16*	RM	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in *xmm1*.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW *ymm1, xmm2/m16*	RM	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in *ymm1*.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD *xmm1, xmm2/m32*	RM	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in *xmm1*.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD *ymm1, xmm2/m32*	RM	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in *ymm1*.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ *xmm1, xmm2/m64*	RM	V/V	AVX2	Broadcast a qword element in mem to two locations in *xmm1*.
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ *ymm1, xmm2/m64*	RM	V/V	AVX2	Broadcast a qword element in mem to four locations in *ymm1*.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 *ymm1, m128*	RM	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits 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

Load integer data from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

The destination operand is a YMM register. The source operand is 8-bit, 16-bit 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. VPBROADCASTB/D/W/Q also support XMM register as the source operand.

VBROADCASTI128: The destination operand is a YMM register. The source operand is 128-bit memory location. Register source encodings for VBROADCASTI128 are reserved and will #UD.

VPBROADCASTB/W/D/Q is supported in both 128-bit and 256-bit wide versions.

VBROADCASTI128 is only supported as a 256-bit wide version.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. Attempts to execute any VPBROADCAST* instruction with VEX.W = 1 will cause #UD. If VBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

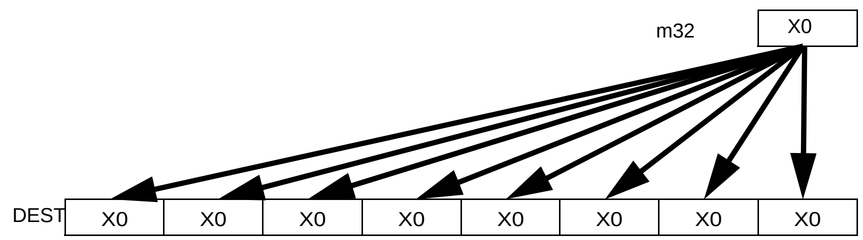

Figure 4-33. VPBROADCASTD Operation (VEX.256 encoded version)

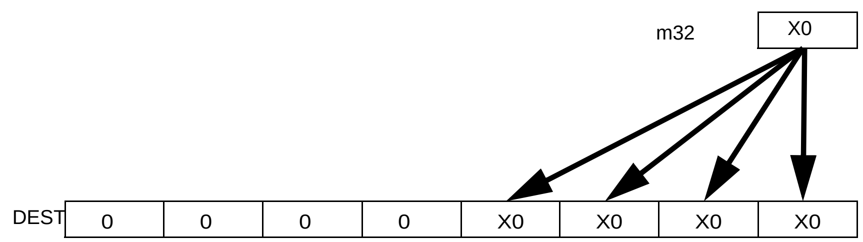

Figure 4-34. VPBROADCASTD Operation (128-bit version)

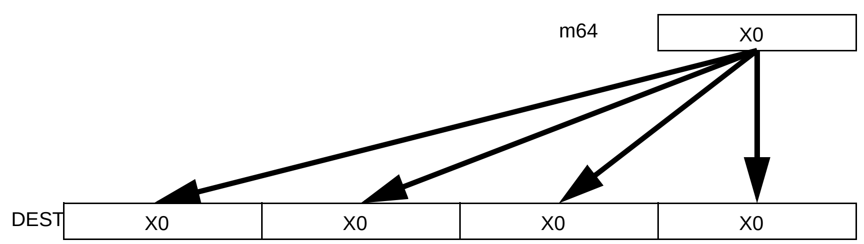

Figure 4-35. VPBROADCASTQ Operation

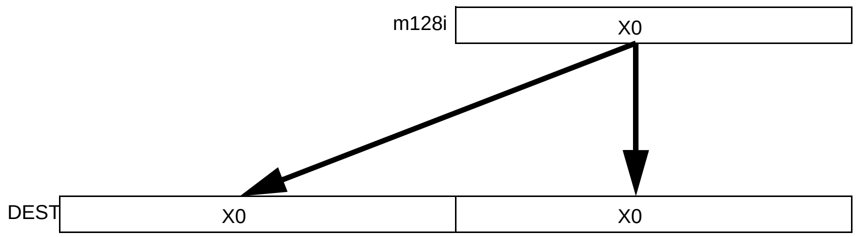

Figure 4-36. VBROADCASTI128 Operation

Operation

VPBROADCASTB (VEX.128 encoded version)

temp ← SRC[7:0]
FOR j ← 0 TO 15
DEST[7+j*8: j*8] ← temp
ENDFOR
DEST[VLMAX-1:128] ← 0

VPBROADCASTB (VEX.256 encoded version)

temp ← SRC[7:0]
FOR j ← 0 TO 31
DEST[7+j*8: j*8] ← temp
ENDFOR

VPBROADCASTW (VEX.128 encoded version)

temp ← SRC[15:0]
FOR j ← 0 TO 7
DEST[15+j*16: j*16] ← temp
ENDFOR
DEST[VLMAX-1:128] ← 0

VPBROADCASTW (VEX.256 encoded version)

temp ← SRC[15:0]
FOR j ← 0 TO 15
DEST[15+j*16: j*16] ← temp
ENDFOR

VPBROADCASTD (128 bit version)

temp ← SRC[31:0]
FOR j ← 0 TO 3
DEST[31+j*32: j*32] ← temp
ENDFOR
DEST[VLMAX-1:128] ← 0

VPBROADCASTD (VEX.256 encoded version)

```
temp ← SRC[31:0]
FOR j ← 0 TO 7
DEST[31+j*32: j*32] ← temp
ENDFOR
```

VPBROADCASTQ (VEX.128 encoded version)

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

VPBROADCASTQ (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
```

VBROADCASTI128

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

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB: __m256i _mm256_broadcastb_epi8(__m128i);

VPBROADCASTW: __m256i _mm256_broadcastw_epi16(__m128i);

VPBROADCASTD: __m256i _mm256_broadcastd_epi32(__m128i);

VPBROADCASTQ: __m256i _mm256_broadcastq_epi64(__m128i);

VPBROADCASTB: __m128i _mm_broadcastb_epi8(__m128i);

VPBROADCASTW: __m128i _mm_broadcastw_epi16(__m128i);

VPBROADCASTD: __m128i _mm_broadcastd_epi32(__m128i);

VPBROADCASTQ: __m128i _mm_broadcastq_epi64(__m128i);

VBROADCASTI128: __m256i _mm256_broadcastsi128_si256(__m128i);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.W = 1,
	If VEX.L = 0 for VBROADCASTI128.

VPERMD — Full Doublewords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Permute doublewords in *ymm3/m256* using indexes in *ymm2* and store the result in *ymm1*.

Instruction Operand Encoding

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

Description

Use the index values in each dword element of the first source operand (the second operand) to select a dword element in the second source operand (the third operand), the resultant dword value from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMD encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMD: __m256i _mm256_permutevar8x32_epi32(__m256i a, __m256i offsets);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0 for VPERMD, If VEX.W = 1.

VPERMPD — Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX2	Permute double-precision floating-point elements in *ymm2/m256* using indexes in *imm8* and store the result in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Use two-bit index values in the immediate byte to select a double-precision floating-point element in the source operand; the resultant data from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple location in the destination operand.

An attempt to execute VPERMPD encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (VEX.256 encoded version)

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD: __m256d _mm256_permute4x64_pd(__m256d a, int control) ;

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0.

VPERMPS — Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 16 /r VPERMPS *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Permute single-precision floating-point elements in *ymm3/m256* using indexes in *ymm2* and store the result in *ymm1*.

Instruction Operand Encoding

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

Description

Use the index values in each dword element of the first source operand (the second operand) to select a single-precision floating-point element in the second source operand (the third operand), the resultant data from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMPS encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPS (VEX.256 encoded version)

```
DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS: __m256i _mm256_permutevar8x32_ps(__m256 a, __m256i offsets)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VPERMQ — Qwords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX2	Permute qwords in *ymm2/m256* using indexes in *imm8* and store the result in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Use two-bit index values in the immediate byte to select a qword element in the source operand, the resultant qword value from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

An attempt to execute VPERMQ encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMQ (VEX.256 encoded version)

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMQ: __m256i _mm256_permute4x64_epi64(__m256i a, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0.

VPERM2I128 — Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 46 /r ib VPERM2I128 *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Permute 128-bit integer data in *ymm2* and *ymm3/mem* using controls from *imm8* and store result in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

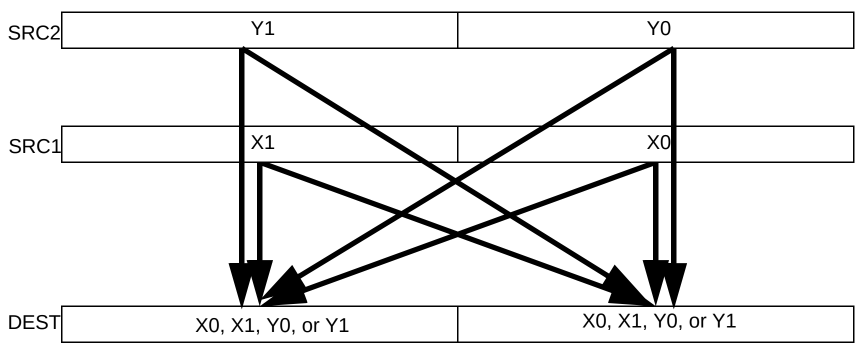

Figure 4-37. VPERM2I128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2I128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC
CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VPERMILPD — Permute Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Permute double-precision floating-point values in *xmm2* using controls from *xmm3/mem* and store result in *xmm1*.
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Permute double-precision floating-point values in *ymm2* using controls from *ymm3/mem* and store result in *ymm1*.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Permute double-precision floating-point values in *xmm2/mem* using controls from *imm8*.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX	Permute double-precision floating-point values in *ymm2/mem* using controls from *imm8*.

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
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Permute double-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of the second source operand (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

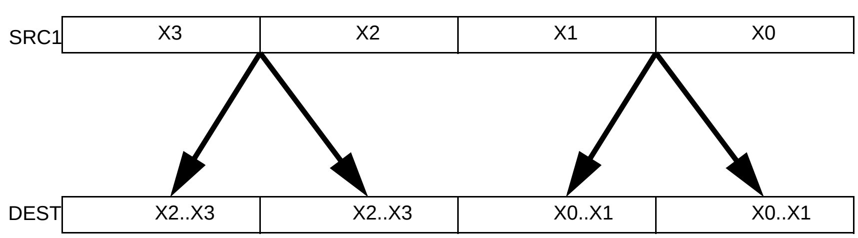

Figure 4-38. VPERMILPD operation

There is one control byte per destination double-precision element. Each control byte is aligned with the low 8 bits of the corresponding double-precision destination element. Each control byte contains a 1-bit select field (see Figure 4-39) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

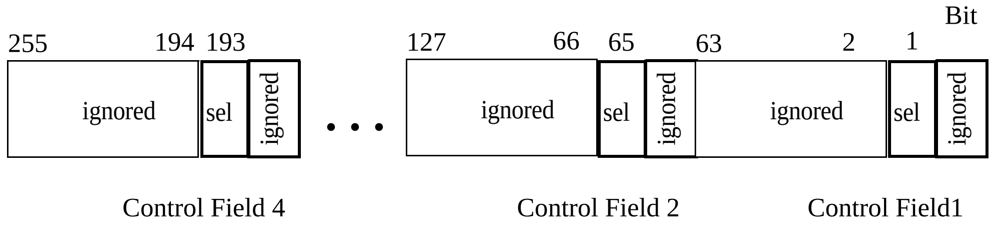

Figure 4-39. VPERMILPD Shuffle Control

(immediate control version)

Permute double-precision floating-point values in the first source operand (second operand) using two, 1-bit control fields in the low 2 bits of the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register.

Note: For the VEX.128.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (256-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64]←SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128]←SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128]←SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192]←SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192]←SRC1[255:192]
```

VPERMILPD (128-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64]←SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

VPERMILPD (256-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64]←SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128]←SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128]←SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192]←SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192]←SRC1[255:192]
```

VPERMILPD (128-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64]←SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPD: __m128d _mm_permute_pd (__m128d a, int control)

VPERMILPD: __m256d _mm256_permute_pd (__m256d a, int control)

VPERMILPD: __m128d _mm_permutevar_pd (__m128d a, __m128i control);

VPERMILPD: __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.W = 1

VPERMILPS — Permute Single-Precision Floating-Point Values

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Permute single-precision floating-point values in *xmm2* using controls from *xmm3/mem* and store result in *xmm1*.
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Permute single-precision floating-point values in *xmm2/mem* using controls from *imm8* and store result in *xmm1*.
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Permute single-precision floating-point values in *ymm2* using controls from *ymm3/mem* and store result in *ymm1*.
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX	Permute single-precision floating-point values in *ymm2/mem* using controls from *imm8* and store result in *ymm1*.

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
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

(variable control version)

Permute single-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of corresponding elements the shuffle control (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

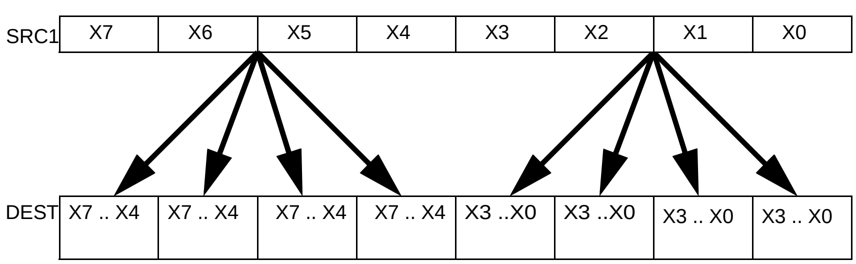

Figure 4-40. VPERMILPS Operation

There is one control byte per destination single-precision element. Each control byte is aligned with the low 8 bits of the corresponding single-precision destination element. Each control byte contains a 2-bit select field (see Figure 4-41) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

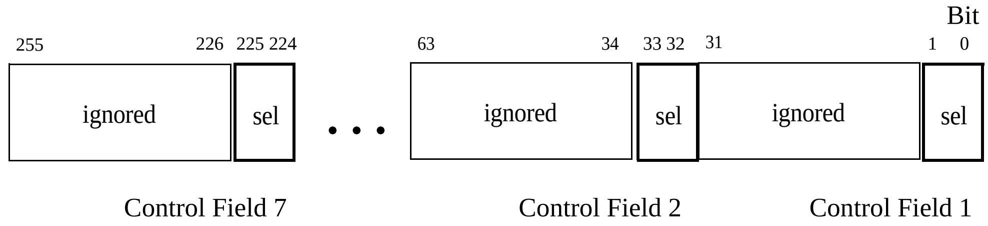

Figure 4-41. VPERMILPS Shuffle Control

(immediate control version)

Permute single-precision floating-point values in the first source operand (second operand) using four 2-bit control fields in the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register. This is similar to a wider version of PSHUFD, just operating on single-precision floating-point values.

Note: For the VEX.128.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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
}
```

VPERMILPS (256-bit immediate 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(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[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(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate 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(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[VLMAX-1:128] ← 0
```

VPERMILPS (256-bit variable version)

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ← Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ← Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ← Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);

VPERMILPS (128-bit variable version)

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS: __m128 _mm_permute_ps (__m128 a, int control);

VPERMILPS: __m256 _mm256_permute_ps (__m256 a, int control);

VPERMILPS: __m128 _mm_permutevar_ps (__m128 a, __m128i control);

VPERMILPS: __m256 _mm256_permutevar_ps (__m256 a, __m256i control);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.W = 1.

VPERM2F128 — Permute Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX	Permute 128-bit floating-point fields in *ymm2* and *ymm3/mem* using controls from *imm8* and store result in *ymm1*.

Instruction Operand Encoding

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

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

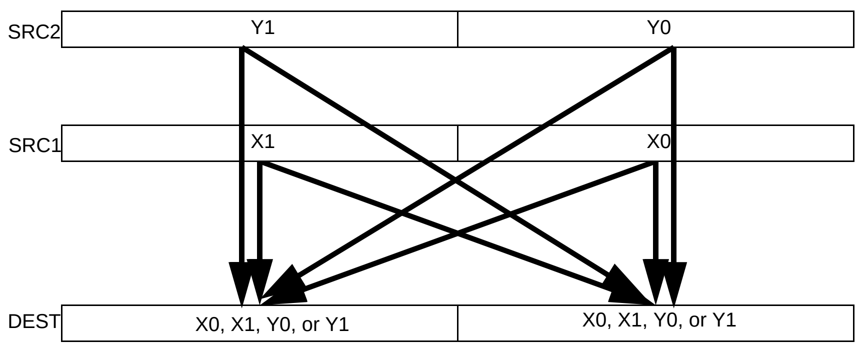

Figure 4-42. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

```
VPERM2F128
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)

VPERM2F128: __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)

VPERM2F128: __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0
	If VEX.W = 1.

VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOVD *xmm1, xmm2, m128*	RVM	V/V	AVX2	Conditionally load dword values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD *ymm1, ymm2, m256*	RVM	V/V	AVX2	Conditionally load dword values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ *xmm1, xmm2, m128*	RVM	V/V	AVX2	Conditionally load qword values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVQ *ymm1, ymm2, m256*	RVM	V/V	AVX2	Conditionally load qword values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD *m128, xmm1, xmm2*	MVR	V/V	AVX2	Conditionally store dword values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVD *m256, ymm1, ymm2*	MVR	V/V	AVX2	Conditionally store dword values from *ymm2* using mask in *ymm1*.
VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ *m128, xmm1, xmm2*	MVR	V/V	AVX2	Conditionally store qword values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVQ *m256, ymm1, ymm2*	MVR	V/V	AVX2	Conditionally store qword values from *ymm2* using mask in *ymm1*.

Instruction Operand Encoding

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

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VPMASKMOVD - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VPMASKMOVD -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VPMASKMOVQ - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VPMASKMOVQ - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VPMASKMOVD - 256-bit store

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

VPMASKMOVD - 128-bit store

IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VPMASKMOVQ - 256-bit store

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

VPMASKMOVQ - 128-bit store

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)

VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)

VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);

VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);

VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)

VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)

VPMASKMOVQ: __m128i _mm_maskload_epi64(__int cont *a, __m128i mask);

VPMASKMOVQ: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations).

VPSLLVD/VPSLLVQ — Variable Bit Shift Left Logical

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX2	Shift bits in doublewords in *xmm2* left by amount specified in the corresponding element of *xmm3/m128* while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX2	Shift bits in quadwords in *xmm2* left by amount specified in the corresponding element of *xmm3/m128* while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shift bits in doublewords in *ymm2* left by amount specified in the corresponding element of *ymm3/m256* while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shift bits in quadwords in *ymm2* left by amount specified in the corresponding element of *ymm3/m256* while shifting in 0s.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

Operation

VPSLLVD (VEX.128 version)

```
COUNT_0 ← SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ← 0;
```

VPSLLVQ (VEX.128 version)

```
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;
```

VPSLLVQ (VEX.256 version)

```
COUNT_0 ← SRC2[5 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVD: __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)

VPSLLVD: __m128i _mm_sllv_epi32 (__m128i m, __m128i count)

VPSLLVQ: __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

VPSLLVQ: __m128i _mm_sllv_epi64 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VPSRAVD — Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX2	Shift bits in doublewords in *xmm2* right by amount specified in the corresponding element of *xmm3/m128* while shifting in the sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shift bits in doublewords in *ymm2* right by amount specified in the corresponding element of *ymm3/m256* while shifting in the sign bits.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual doubleword data elements in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in each data element are shifted right, the empty high-order bits are filled with the sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31, then the destination data element are filled with the corresponding sign bit of the source element.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

Operation

VPSRAVD (VEX.128 version)

```
COUNT_0 ← SRC2[31: 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 112];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31] );
FI;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
ELSE
    For (i = 0 to 31) DEST[i + 96] ← (SRC1[127] );
FI;
DEST[VLMAX-1:128] ← 0;
```

VPSRAVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31] );
FI;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
ELSE
    For (i = 0 to 31) DEST[i + 224] ← (SRC1[255] );
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD: __m256i _mm256_srav_epi32 (__m256i m, __m256i count)

VPSRAVD: __m128i _mm_srav_epi32 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

VPSRLVD/VPSRLVQ — Variable Bit Shift Right Logical

Opcode/ Instruction	Op/ EN	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX2	Shift bits in doublewords in *xmm2* right by amount specified in the corresponding element of *xmm3/m128* while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX2	Shift bits in quadwords in *xmm2* right by amount specified in the corresponding element of *xmm3/m128* while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shift bits in doublewords in *ymm2* right by amount specified in the corresponding element of *ymm3/m256* while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Shift bits in quadwords in *ymm2* right by amount specified in the corresponding element of *ymm3/m256* while shifting in 0s.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory location.

Operation

VPSRLVD (VEX.128 version)

```
COUNT_0 ← SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX-1:128] ← 0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
DEST[255:224] ← 0;
```

VPSRLVQ (VEX.128 version)

```
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
IF COUNT_1 < 64THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
DEST[127:64] ← 0;
DEST[VLMAX-1:128] ← 0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 ← SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVD: __m256i _mm256_srlv_epi32 (__m256i m, __m256i count);

VPSRLVD: __m128i _mm_srlv_epi32 (__m128i m, __m128i count);

VPSRLVQ: __m256i _mm256_srlv_epi64 (__m256i m, __m256i count);

VPSRLVQ: __m128i _mm_srlv_epi64 (__m128i m, __m128i count);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS *xmm1, xmm2/m128*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.256.66.0F38.W0 0E /r VTESTPS *ymm1, ymm2/m256*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.128.66.0F38.W0 0F /r VTESTPD *xmm1, xmm2/m128*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.
VEX.256.66.0F38.W0 0F /r VTESTPD *ymm1, ymm2/m256*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point 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

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

128-bit version: 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

VTESTPS (128-bit version)

```
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPS (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPD (128-bit version)

```
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPD (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VTESTPS

int _mm256_testz_ps (__m256 s1, __m256 s2);

int _mm256_testc_ps (__m256 s1, __m256 s2);

int _mm256_testnzc_ps (__m256 s1, __m128 s2);

int _mm_testz_ps (__m128 s1, __m128 s2);

int _mm_testc_ps (__m128 s1, __m128 s2);

int _mm_testnzc_ps (__m128 s1, __m128 s2);

VTESTPD

int _mm256_testz_pd (__m256d s1, __m256d s2);

int _mm256_testc_pd (__m256d s1, __m256d s2);

int _mm256_testnzc_pd (__m256d s1, __m256d s2);

int _mm_testz_pd (__m128d s1, __m128d s2);

int _mm_testc_pd (__m128d s1, __m128d s2);

int _mm_testnzc_pd (__m128d s1, __m128d 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.
	If VEX.W = 1 for VTESTPS or VTESTPD.

VZEROALL—Zero All YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.0F.WIG 77 VZEROALL	NP	V/V	AVX	Zero all YMM registers.

Instruction Operand Encoding

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

Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROALL (VEX.256 encoded version)

```
IF (64-bit mode)
    YMM0[VLMAX-1:0] ← 0
    YMM1[VLMAX-1:0] ← 0
    YMM2[VLMAX-1:0] ← 0
    YMM3[VLMAX-1:0] ← 0
    YMM4[VLMAX-1:0] ← 0
    YMM5[VLMAX-1:0] ← 0
    YMM6[VLMAX-1:0] ← 0
    YMM7[VLMAX-1:0] ← 0
    YMM8[VLMAX-1:0] ← 0
    YMM9[VLMAX-1:0] ← 0
    YMM10[VLMAX-1:0] ← 0
    YMM11[VLMAX-1:0] ← 0
    YMM12[VLMAX-1:0] ← 0
    YMM13[VLMAX-1:0] ← 0
    YMM14[VLMAX-1:0] ← 0
    YMM15[VLMAX-1:0] ← 0
ELSE
    YMM0[VLMAX-1:0] ← 0
    YMM1[VLMAX-1:0] ← 0
    YMM2[VLMAX-1:0] ← 0
    YMM3[VLMAX-1:0] ← 0
    YMM4[VLMAX-1:0] ← 0
    YMM5[VLMAX-1:0] ← 0
    YMM6[VLMAX-1:0] ← 0
    YMM7[VLMAX-1:0] ← 0
    YMM8-15: Unmodified
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL: _mm256_zeroall()

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

VZEROUPPER—Zero Upper Bits of YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	NP	V/V	AVX	Zero upper 128 bits of all YMM registers.

Instruction Operand Encoding

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

Description

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROUPPER

```
IF (64-bit mode)
    YMM0[VLMAX-1:128] ← 0
    YMM1[VLMAX-1:128] ← 0
    YMM2[VLMAX-1:128] ← 0
    YMM3[VLMAX-1:128] ← 0
    YMM4[VLMAX-1:128] ← 0
    YMM5[VLMAX-1:128] ← 0
    YMM6[VLMAX-1:128] ← 0
    YMM7[VLMAX-1:128] ← 0
    YMM8[VLMAX-1:128] ← 0
    YMM9[VLMAX-1:128] ← 0
    YMM10[VLMAX-1:128] ← 0
    YMM11[VLMAX-1:128] ← 0
    YMM12[VLMAX-1:128] ← 0
    YMM13[VLMAX-1:128] ← 0
    YMM14[VLMAX-1:128] ← 0
    YMM15[VLMAX-1:128] ← 0
ELSE
    YMM0[VLMAX-1:128] ← 0
    YMM1[VLMAX-1:128] ← 0
    YMM2[VLMAX-1:128] ← 0
    YMM3[VLMAX-1:128] ← 0
    YMM4[VLMAX-1:128] ← 0
    YMM5[VLMAX-1:128] ← 0
    YMM6[VLMAX-1:128] ← 0
    YMM7[VLMAX-1:128] ← 0
    YMM8-15: unmodified
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER: _mm256_zeroupper()

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

WAIT/FWAIT—Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9B	WAIT	NP	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	NP	Valid	Valid	Check pending unmasked floating-point exceptions.

Instruction Operand Encoding

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

Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

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

Operation

```
CheckForPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 09	WBINVD	NP	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

Instruction Operand Encoding

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

Description

Writes back all modified cache lines in the processor’s internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

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

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /2 WRFSBASE *r32*	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
REX.W + F3 0F AE /2 WRFSBASE *r64*	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 0F AE /3 WRGSBASE *r32*	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
REX.W + F3 0F AE /3 WRGSBASE *r64*	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

Instruction Operand Encoding

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

Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source 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 register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

Operation

```
FS/GS segment base address ← SRC;
```

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

```
WRFSBASE:    void _writefsbase_u32( unsigned int );
WRFSBASE:    _writefsbase_u64( unsigned __int64 );
WRGSBASE:    void _writegsbase_u32( unsigned int );
WRGSBASE:    _writegsbase_u64( unsigned __int64 );
```

Protected Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The WRFSBASE and WRGSBASE 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
#GP(0)	If the source register contains a non-canonical address.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	NP	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

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

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 35, "Model-Specific Registers (MSRs)", in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

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 WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

MSR[ECX] ← EDX:EAX;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRPKRU—Write Data to User Page Key Register

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
0F 01 EF	WRPKRU	NP	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

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

Description

The WRPKRU instruction loads the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, a general-protection exception (#GP) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

In 64-bit mode, WRPKRU ignores bits 63:32 of each of RAX, RCX, and RDX.

Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
	If EDX ≠ 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.

XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE[1]	A hint used with an “XACQUIRE-enabled“ instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an “XRELEASE-enabled“ instruction to end lock elision on the instruction memory operand address.

NOTES:

1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, Section 15.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,

- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE

```
IF XACQUIRE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
            HLE_NEST_COUNT++
            IF (HLE_NEST_COUNT = 1) THEN
                HLE_ACTIVE ← 1
                IF 64-bit mode
                    THEN
                        restartRIP ← instruction pointer of the XACQUIRE-enabled instruction
                    ELSE
                        restartEIP ← instruction pointer of the XACQUIRE-enabled instruction
                FI;
                Enter HLE Execution (* record register state, start tracking memory state *)
            FI; (* HLE_NEST_COUNT = 1*)
            IF ElisionBufferAvailable
                THEN
                    Allocate elision buffer
                    Record address and data for forwarding and commit checking
                    Perform elision
                ELSE
                    Perform lock acquire operation transactionally but without elision
            FI;
        ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
                GOTO HLE_ABORT_PROCESSING
        FI;
    ELSE
        Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
FI;
```

XRELEASE

```
IF XRELEASE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT > 0)
            THEN
                HLE_NEST_COUNT--
                IF lock address matches in elision buffer THEN
                    IF lock satisfies address and value requirements THEN
                        Deallocate elision buffer
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI;
                IF (HLE_NEST_COUNT = 0)
                    THEN
                        IF NoAllocatedElisionBuffer
                            THEN
```

```
                                        Try to commit transactional execution
                                        IF fail to commit transactional execution
                                            THEN
                                                GOTO HLE_ABORT_PROCESSING;
                                            ELSE (* commit success *)
                                                HLE_ACTIVE ← 0
                                        FI;
                                    ELSE
                                        GOTO HLE_ABORT_PROCESSING
                                FI;
                        FI;
                FI; (* HLE_NEST_COUNT > 0 *)
    ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE ← 0
    HLE_NEST_COUNT ← 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP ← restartRIP
        ELSE
            EIP ← restartEIP
    FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
```

SIMD Floating-Point Exceptions

None

Other Exceptions

#GP(0)	If the use of prefix causes instruction length to exceed 15 bytes.

XABORT — Transactional Abort

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

Operation

XABORT

```
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XABORT: void _xabort(unsigned int);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK prefix is used.

XADD—Exchange and Add

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C0 /*r*	XADD *r/m8, r8*	MR	Valid	Valid	Exchange *r8* and *r/m8*; load sum into *r/m8*.
REX + 0F C0 /*r*	XADD *r/m8**, *r8**	MR	Valid	N.E.	Exchange *r8* and *r/m8*; load sum into *r/m8*.
0F C1 /*r*	XADD *r/m16, r16*	MR	Valid	Valid	Exchange *r16* and *r/m16*; load sum into *r/m16*.
0F C1 /*r*	XADD *r/m32, r32*	MR	Valid	Valid	Exchange *r32* and *r/m32*; load sum into *r/m32*.
REX.W + 0F C1 /*r*	XADD *r/m64, r64*	MR	Valid	N.E.	Exchange *r64* and *r/m64*; load sum into *r/m64*.

NOTES:

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

Instruction Operand Encoding

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

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

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

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

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

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

XBEGIN — Transactional Begin

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

Operation

XBEGIN

```
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                        fallbackRIP ← RIP + SignExtend64(IMM)
                                    (* RIP is instruction following XBEGIN instruction *)
                    ELSE
                        fallbackEIP ← EIP + SignExtend32(IMM)
                                    (* EIP is instruction following XBEGIN instruction *)
            FI;

            IF (64-bit mode)
                THEN IF (fallbackRIP is not canonical)
                    THEN #GP(0)
                FI;
                ELSE IF (fallbackEIP outside code segment limit)
                    THEN #GP(0)
                FI;
            FI;

            RTM_ACTIVE ← 1
            Enter RTM Execution (* record register state, start tracking memory state*)
```

```
        FI; (* RTM_NEST_COUNT = 1 *)
    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
        GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN: unsigned int _xbegin(void);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.
#GP(0)	If the fallback address is outside the CS segment.

Real-Address Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK prefix is used.

#GP(0)	If the fallback address is non-canonical.

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90+*rw*	XCHG AX, *r16*	O	Valid	Valid	Exchange *r16* with AX.
90+*rw*	XCHG *r16*, AX	O	Valid	Valid	Exchange AX with *r16.*
90+*rd*	XCHG EAX, *r32*	O	Valid	Valid	Exchange *r32* with EAX.
REX.W + 90+*rd*	XCHG RAX, *r64*	O	Valid	N.E.	Exchange *r64* with RAX.
90+*rd*	XCHG *r32*, EAX	O	Valid	Valid	Exchange EAX with *r32.*
REX.W + 90+*rd*	XCHG *r64*, RAX	O	Valid	N.E.	Exchange RAX with *r64.*
86 /*r*	XCHG *r/m8, r8*	MR	Valid	Valid	Exchange *r8* (byte register) with byte from *r/m8.*
REX + 86 /*r*	XCHG *r/m8**, r8**	MR	Valid	N.E.	Exchange *r8* (byte register) with byte from *r/m8.*
86 /*r*	XCHG *r8, r/m8*	RM	Valid	Valid	Exchange byte from *r/m8* with *r8* (byte register).
REX + 86 /*r*	XCHG *r8**, r/m8**	RM	Valid	N.E.	Exchange byte from *r/m8* with *r8* (byte register).
87 /*r*	XCHG *r/m16, r16*	MR	Valid	Valid	Exchange *r16* with word from *r/m16.*
87 /*r*	XCHG *r16, r/m16*	RM	Valid	Valid	Exchange word from *r/m16* with *r16.*
87 /*r*	XCHG *r/m32, r32*	MR	Valid	Valid	Exchange *r32* with doubleword from *r/m32.*
REX.W + 87 /*r*	XCHG *r/m64, r64*	MR	Valid	N.E.	Exchange *r64* with quadword from *r/m64.*
87 /*r*	XCHG *r32, r/m32*	RM	Valid	Valid	Exchange doubleword from *r/m32* with *r32.*
REX.W + 87 /*r*	XCHG *r64, r/m64*	RM	Valid	N.E.	Exchange quadword from *r/m64* with *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	AX/EAX/RAX (r, w)	opcode + rd (r, w)	NA	NA
O	opcode + rd (r, w)	AX/EAX/RAX (r, w)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 3A*, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

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

Operation

TEMP ← DEST;
DEST ← SRC;
SRC ← TEMP;

Flags Affected

None.

Protected Mode Exceptions

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

XEND — Transactional End

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
0F 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE ← 0
        FI;
    FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XEND: void _xend(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK or 66H or F2H or F3H prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 D0	XGETBV	NP	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

Instruction Operand Encoding

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

Description

Reads the contents of the extended control register (XCR) 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 XCR 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 XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

Operation

EDX:EAX ← XCR[ECX];

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XGETBV: unsigned __int64 _xgetbv(unsigned int);

Protected Mode Exceptions

#GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions

#GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
If 66H, F3H or F2H 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.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT *m8*	NP	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	NP	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	NP	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

Instruction Operand Encoding

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

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operand" form and the "no-operand" form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table's base address. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
    THEN
        AL ← (DS:BX + ZeroExtend(AL));
    ELSE IF (AddressSize = 32)
        AL ← (DS:EBX + ZeroExtend(AL)); FI;
    ELSE (AddressSize = 64)
        AL ← (RBX + ZeroExtend(AL));
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

#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.

XOR—Logical Exclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
34 *ib*	XOR AL, *imm8*	I	Valid	Valid	AL XOR *imm8.*
35 *iw*	XOR AX, *imm16*	I	Valid	Valid	AX XOR *imm16.*
35 *id*	XOR EAX, *imm32*	I	Valid	Valid	EAX XOR *imm32.*
REX.W + 35 *id*	XOR RAX, *imm32*	I	Valid	N.E.	RAX XOR *imm32 (sign-extended).*
80 /6 *ib*	XOR *r/m8, imm8*	MI	Valid	Valid	*r/m8* XOR *imm8.*
REX + 80 /6 *ib*	XOR *r/m8*, imm8*	MI	Valid	N.E.	*r/m8* XOR *imm8.*
81 /6 *iw*	XOR *r/m16, imm16*	MI	Valid	Valid	*r/m16* XOR *imm16.*
81 /6 *id*	XOR *r/m32, imm32*	MI	Valid	Valid	*r/m32* XOR *imm32.*
REX.W + 81 /6 *id*	XOR *r/m64, imm32*	MI	Valid	N.E.	*r/m64* XOR *imm32 (sign-extended).*
83 /6 *ib*	XOR *r/m16, imm8*	MI	Valid	Valid	*r/m16* XOR *imm8 (sign-extended).*
83 /6 *ib*	XOR *r/m32, imm8*	MI	Valid	Valid	*r/m32* XOR *imm8 (sign-extended).*
REX.W + 83 /6 *ib*	XOR *r/m64, imm8*	MI	Valid	N.E.	*r/m64* XOR *imm8 (sign-extended).*
30 /*r*	XOR *r/m8, r8*	MR	Valid	Valid	*r/m8* XOR *r8.*
REX + 30 /*r*	XOR *r/m8*, r8**	MR	Valid	N.E.	*r/m8* XOR *r8.*
31 /*r*	XOR *r/m16, r16*	MR	Valid	Valid	*r/m16* XOR *r16.*
31 /*r*	XOR *r/m32, r32*	MR	Valid	Valid	*r/m32* XOR *r32.*
REX.W + 31 /*r*	XOR *r/m64, r64*	MR	Valid	N.E.	*r/m64* XOR *r64.*
32 /*r*	XOR *r8, r/m8*	RM	Valid	Valid	*r8* XOR *r/m8.*
REX + 32 /*r*	XOR *r8*, r/m8**	RM	Valid	N.E.	*r8* XOR *r/m8.*
33 /*r*	XOR *r16, r/m16*	RM	Valid	Valid	*r16* XOR *r/m16.*
33 /*r*	XOR *r32, r/m32*	RM	Valid	Valid	*r32* XOR *r/m32.*
REX.W + 33 /*r*	XOR *r64, r/m64*	RM	Valid	N.E.	*r64* XOR *r/m64.*

NOTES:

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

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 exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

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

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

Operation

DEST ← DEST XOR SRC;

Flags Affected

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

Protected Mode Exceptions

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

XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

Opcode/ Instruction	**Op/ En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
66 0F 57 /*r* XORPD *xmm1*, *xmm2/m128*	RM	V/V	SSE2	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG 57 /r VXORPD *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in *xmm2* and *xmm3/mem*.
VEX.NDS.256.66.0F.WIG 57 /r VXORPD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in *ymm2* and *ymm3/mem*.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

XORPD (128-bit Legacy SSE version)

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

VXORPD (VEX.128 encoded version)

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

VXORPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

XORPD: __m128d _mm_xor_pd(__m128d a, __m128d b)

VXORPD: __m256d _mm256_xor_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 57 /r XORPS *xmm1, xmm2/m128*	RM	V/V	SSE	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.0F.WIG 57 /r VXORPS *xmm1,xmm2, xmm3/m128*	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in *xmm2* and *xmm3/mem*.
VEX.NDS.256.0F.WIG 57 /r VXORPS *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in *ymm2* and *ymm3/mem*.

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

XORPS (128-bit Legacy SSE version)

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

VXORPS (VEX.128 encoded version)

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

VXORPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

XORPS: __m128 _mm_xor_ps(__m128 a, __m128 b)

VXORPS: __m256 _mm256_xor_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AE /5	XRSTOR *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
REX.W+ 0F AE /5	XRSTOR64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.8, “Operation of XRSTOR,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[*i*] = 0, XRSTOR does not update state component *i*.[1]
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;    /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
IF in VMX non-root operation
    THEN VMXNR ← 1;
```

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

```
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        If RFBM[0] = 1
            THEN
                IF RSTORMASK[0] = 1
                    THEN load x87 state from legacy region of XSAVE area;
                    ELSE initialize x87 state;
                FI;
        FI;
        If RFBM[1] = 1
            THEN
                IF RSTORMASK[1] = 1
                    THEN load XMM registers from legacy region of XSAVE area;
                    ELSE set all XMM registers to 0;
                FI;
        FI;
        If RFBM[2] = 1
            THEN
                IF RSTORMASK[2] = 1
                    THEN load AVX state from extended region (standard format) of XSAVE area;
                    ELSE initialize AVX state;
                FI;
        FI;
        If RFBM[1] = 1 or RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;
FI;
    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN    /* compacted form not supported */
                #GP(0);
        FI;
        If RFBM[0] = 1
            THEN
                IF RSTORMASK[0] = 1
                    THEN load x87 state from legacy region of XSAVE area;
                    ELSE initialize x87 state;
                FI;
        FI;
        If RFBM[1] = 1
            THEN
                IF RSTORMASK[1] = 1
                    THEN load SSE state from legacy region of XSAVE area;
                    ELSE initialize SSE state;
                FI;
        FI;
        If RFBM[2] = 1
            THEN
```

```
                IF RSTORMASK[2] = 1
                    THEN load AVX state from extended region (compacted format) of XSAVE area;
                    ELSE initialize AVX state;
                FI;
        FI;
FI;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: void _xrstor(void * , unsigned __int64);

XRSTOR: void _xrstor64(void * , unsigned __int64);

Protected Mode Exceptions

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

If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.

If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.

If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

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

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

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

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

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

If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes 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

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /3	XRSTORS *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
REX.W+ 0F C7 /3	XRSTORS64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.12, "Operation of XRSTORS," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area").
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
```

```
LAXA ← linear address of XSAVE area;

If RFBM[0] = 1
    THEN
        IF RSTORMASK[0] = 1
            THEN load x87 state from legacy region of XSAVE area;
            ELSE initialize x87 state;
        FI;
FI;
If RFBM[1] = 1
    THEN
        IF RSTORMASK[1] = 1
            THEN load SSE state from legacy region of XSAVE area;
            ELSE initialize SSE state;
        FI;
FI;
If RFBM[2] = 1
    THEN
        IF RSTORMASK[2] = 1
            THEN load AVX state from extended region (compacted format) of XSAVE area;
            ELSE initialize AVX state;
        FI;
FI;
XRSTOR_INFO ← 〈CPL,VMXNR,LAXA,COMPMASK〉;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS: void _xrstors(void * , unsigned __int64);

XRSTORS64: void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check

exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes 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

#GP(0)	If CPL > 0.
	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protec-

tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVE—Save Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AE /4	XSAVE *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem*.
REX.W+ 0F AE /4	XSAVE64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem*.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.7, "Operation of XSAVE," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if RFBM[*i*] = 1.[1]
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") If RFBM[*i*] = 0, XSAVE writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;
IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area;
FI;
```

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

```
IF RFBM[2] = 1
    THEN store AVX state into extended region of XSAVE area;
FI;
IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

XSTATE_BV field in XSAVE header ← (OLD_BV AND ~RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVE: void _xsave(void * , unsigned __int64);

XSAVE: void _xsave64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes 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

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States with Compaction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /4	XSAVEC *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem* with compaction.
REX.W+ 0F C7 /4	XSAVEC64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem* with compaction.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.10, "Operation of XSAVEC," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.")
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2,3] (See Section 13.4.2, "XSAVE Header.") XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
COMPMASK ← RFBM OR 80000000_00000000H;

IF RFBM[0] = 1 and XINUSE[0] = 1
```

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

```
    THEN store x87 state into legacy region of XSAVE area;
FI;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
FI;

XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;1
XCOMP_BV field in XSAVE header ← COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEC: void _xsavec(void * , unsigned __int64);

XSAVEC64: void _xsavec64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.

1. If MXCSR does not have its initial value of 1F80H, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1, regardless of the value of XINUSE[1].

Virtual-8086 Mode Exceptions

Same exceptions as in protected 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 a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F AE /6 XSAVEOPT *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.
REX.W + 0F AE /6 XSAVEOPT64 *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.9, "Operation of XSAVEOPT," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it uses compaction and that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVEOPT writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. If RFBM[*i*] = 0, XSAVEOPT writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;
```

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

```
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← 00000000_00000000H;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,COMPMASK⟩
    THEN MODOPT ← 1;
    ELSE MODOPT ← 0;
FI;

IF RFBM[0] = 1 and XINUSE[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
    /* might avoid saving if x87 state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 and XINUSE[1]
    THEN store XMM registers into legacy region of XSAVE area;
    /* might avoid saving if XMM registers are not modified and MODOPT = 1 */
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
    /* might avoid saving if AVX state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

XSTATE_BV field in XSAVE header ← (OLD_BV AND ~RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEOPT: void _xsaveopt(void * , unsigned __int64);

XSAVEOPT: void _xsaveopt64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H 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

#SS(0)	If a 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 a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

XSAVES—Save Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /5	XSAVES *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.
REX.W+ 0F C7 /5	XSAVES64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.

Instruction Operand Encoding

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

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.11, "Operation of XSAVES," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2] (See Section 13.4.2, "XSAVE Header.") XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
IF XRSTOR_INFO = 〈CPL,VMXNR,LAXA,COMPMASK〉
    THEN MODOPT ← 1;
    ELSE MODOPT ← 0;
FI;

IF RFBM[0] = 1 and XINUSE[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
    /* might avoid saving if x87 state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
    /* might avoid saving if SSE state is not modified and MODOPT = 1 */
FI;
IF RFBM[2] = 1 AND XINUSE[2] = 1
    THEN store AVX state into extended region of XSAVE area;
    /* might avoid saving if AVX state is not modified and MODOPT = 1 */
FI;

XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;[1]
XCOMP_BV field in XSAVE header ← COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVES: void _xsaves(void * , unsigned __int64);

XSAVES64: void _xsaves64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check

1. If MXCSR does not have its initial value of 1F80H, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1, regardless of the value of XINUSE[1].

exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes 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

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSETBV—Set Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 D1	XSETBV	NP	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

Instruction Operand Encoding

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

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, "Enabling the XSAVE Feature Set and XSAVE-Enabled Features," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

XCR[ECX] ← EDX:EAX;

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSETBV: void _xsetbv(unsigned int, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The XSETBV 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.

XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
0F 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST: int _xtest(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.
	If LOCK or 66H or F2H or F3H prefix is used.