

# Intel® Architecture Instruction Set Extensions Programming Reference

319433-023

AUGUST 2015

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at [intel.com](http://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 1 FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS

|       |   |     |
|-------|---|-----|
| 1.1   | About This Document.....                                | 1-1 |
| 1.2   | Intel® AVX-512 Instructions Architecture Overview ..... | 1-1 |
| 1.2.1 | 512-Bit Wide SIMD Register Support .....                | 1-2 |
| 1.2.2 | 32 SIMD Register Support .....                          | 1-2 |
| 1.2.3 | Eight Opmask Register Support .....                     | 1-2 |
| 1.2.4 | Instruction Syntax Enhancement .....                    | 1-2 |
| 1.2.5 | EVEX Instruction Encoding Support.....                  | 1-3 |

## CHAPTER 2 INTEL® AVX-512 APPLICATION PROGRAMMING MODEL

|         |   |      |
|---------|---|------|
| 2.1     | Detection of AVX-512 Foundation Instructions .....  | 2-1  |
| 2.2     | Detection of 512-bit Instruction Groups of Intel® AVX-512 Family.....                           | 2-2  |
| 2.3     | Detection of Intel AVX-512 Instruction Groups Operating at 256 and 128-bit Vector Lengths ..... | 2-3  |
| 2.4     | Accessing XMM, YMM AND ZMM Registers .....  | 2-4  |
| 2.5     | Enhanced Vector Programming Environment Using EVEX Encoding.....                                | 2-4  |
| 2.5.1   | OPMASK Register to Predicate Vector Data Processing.....  | 2-5  |
| 2.5.1.1 | Opmask Register KO.....   | 2-6  |
| 2.5.1.2 | Example of Opmask Usages.....   | 2-6  |
| 2.5.2   | OpMask Instructions .....   | 2-7  |
| 2.5.3   | Broadcast.....  | 2-7  |
| 2.5.4   | STATIC ROUNDING MODE AND SUPPRESS ALL EXCEPTIONS .....  | 2-8  |
| 2.5.5   | Compressed Disp8*N Encoding .....   | 2-9  |
| 2.6     | Memory Alignment .....  | 2-10 |
| 2.7     | SIMD Floating-Point Exceptions .....  | 2-11 |
| 2.8     | Instruction Exception Specification .....   | 2-11 |
| 2.9     | CPUID Instruction.....  | 2-12 |
|         | CPUID—CPU Identification.....   | 2-12 |

## CHAPTER 3 SYSTEM PROGRAMMING FOR INTEL® AVX-512

|       |  |     |
|-------|--|-----|
| 3.1   | AVX-512 State, EVEX Prefix and Supported Operating Modes ..... | 3-1 |
| 3.2   | AVX-512 State Management.....                                  | 3-1 |
| 3.2.1 | Detection of ZMM and Opmask State Support.....                 | 3-1 |
| 3.2.2 | Enabling of ZMM and Opmask Register State.....                 | 3-2 |
| 3.2.3 | Enabling of SIMD Floating-Exception Support.....               | 3-3 |
| 3.2.4 | The Layout of XSAVE Sate Save Area .....                       | 3-3 |
| 3.2.5 | XSAVE/XRSTOR Interaction with YMM State and MXCSR .....        | 3-5 |
| 3.2.6 | XSAVE/XRSTOR/XSAVEOPT and Managing ZMM and Opmask States ..... | 3-6 |
| 3.3   | Reset Behavior.....  | 3-7 |
| 3.4   | Emulation.....   | 3-7 |
| 3.5   | Writing floating-point exception handlers .....                | 3-7 |

## CHAPTER 4 AVX-512 INSTRUCTION ENCODING

|       |  |     |
|-------|--|-----|
| 4.1   | Overview Section.....                                  | 4-1 |
| 4.2   | Instruction Format and EVEX.....                       | 4-1 |
| 4.3   | Register Specifier Encoding and EVEX .....             | 4-3 |
| 4.3.1 | Opmask Register Encoding.....                          | 4-4 |
| 4.4   | MAsking support in EVEX .....                          | 4-4 |
| 4.5   | Compressed displacement (disp8*N) support in EVEX..... | 4-5 |
| 4.6   | EVEX encoding of broadcast/Rounding/SAE Support.....   | 4-6 |
| 4.6.1 | Embedded Broadcast Support in EVEX.....                | 4-6 |
| 4.6.2 | Static Rounding Support in EVEX.....                   | 4-6 |
| 4.6.3 | SAE Support in EVEX .....                              | 4-7 |
| 4.6.4 | Vector Length Orthogonality.....                       | 4-7 |
| 4.7   | #UD equations for EVEX.....                            | 4-7 |
| 4.7.1 | State Dependent #UD .....                              | 4-7 |

|         |  |      |
|---------|--|------|
| 4.7.2   | Opcode Independent #UD   | 4-8  |
| 4.7.3   | Opcode Dependent #UD   | 4-8  |
| 4.8     | Device Not Available   | 4-9  |
| 4.9     | Scalar Instructions  | 4-9  |
| 4.10    | Exception Classifications of EVEX-Encoded instructions                           | 4-10 |
| 4.10.1  | Exceptions Type E1 and E1NF of EVEX-Encoded Instructions                         | 4-13 |
| 4.10.2  | Exceptions Type E2 of EVEX-Encoded Instructions                                  | 4-15 |
| 4.10.3  | Exceptions Type E3 and E3NF of EVEX-Encoded Instructions                         | 4-16 |
| 4.10.4  | Exceptions Type E4 and E4NF of EVEX-Encoded Instructions                         | 4-18 |
| 4.10.5  | Exceptions Type E5 and E5NF  | 4-20 |
| 4.10.6  | Exceptions Type E6 and E6NF  | 4-22 |
| 4.10.7  | Exceptions Type E7NM   | 4-24 |
| 4.10.8  | Exceptions Type E9 and E9NF  | 4-25 |
| 4.10.9  | Exceptions Type E10  | 4-27 |
| 4.10.10 | Exception Type E11 (EVEX-only, mem arg no AC, floating-point exceptions)         | 4-29 |
| 4.10.11 | Exception Type E12 and E12NP (VSIB mem arg, no AC, no floating-point exceptions) | 4-30 |
| 4.11    | Exception Classifications of Opmask instructions                                 | 4-32 |

## CHAPTER 5 INSTRUCTION SET REFERENCE, A-Z

|         |  |       |
|---------|--|-------|
| 5.1     | Interpreting Instruction Reference Pages   | 5-1   |
| 5.1.1   | Instruction Format   | 5-1   |
|         | ADDPS—Add Packed Single-Precision Floating-Point Values (THIS IS AN EXAMPLE)                 | 5-2   |
| 5.1.2   | Opcode Column in the Instruction Summary Table   | 5-2   |
| 5.1.3   | Instruction Column in the Instruction Summary Table  | 5-4   |
| 5.1.4   | 64/32 bit Mode Support column in the Instruction Summary Table                               | 5-5   |
| 5.1.5   | CPUID Support column in the Instruction Summary Table  | 5-6   |
| 5.1.5.1 | Operand Encoding Column in the Instruction Summary Table                                     | 5-6   |
| 5.2     | Summary of Terms   | 5-6   |
| 5.3     | Ternary Bit Vector Logic Table   | 5-7   |
| 5.4     | Instruction SET Reference  | 5-9   |
|         | ADDPD—Add Packed Double-Precision Floating-Point Values                                      | 5-10  |
|         | ADDPS—Add Packed Single-Precision Floating-Point Values                                      | 5-13  |
|         | ADDSD—Add Scalar Double-Precision Floating-Point Values                                      | 5-16  |
|         | ADDSS—Add Scalar Single-Precision Floating-Point Values                                      | 5-18  |
|         | VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors  | 5-20  |
|         | VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control                    | 5-23  |
|         | VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control                          | 5-25  |
|         | VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control                        | 5-27  |
|         | ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values                   | 5-29  |
|         | ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values                   | 5-32  |
|         | ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values              | 5-35  |
|         | ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values              | 5-38  |
|         | VBROADCAST—Load with Broadcast Floating-Point Data   | 5-41  |
|         | VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register            | 5-48  |
|         | VPBROADCAST—Load Integer and Broadcast   | 5-51  |
|         | CMPPD—Compare Packed Double-Precision Floating-Point Values                                  | 5-60  |
|         | CMPPS—Compare Packed Single-Precision Floating-Point Values                                  | 5-67  |
|         | CMPSD—Compare Scalar Double-Precision Floating-Point Value                                   | 5-73  |
|         | CMPSS—Compare Scalar Single-Precision Floating-Point Value                                   | 5-77  |
|         | COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS          | 5-82  |
|         | COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS          | 5-84  |
|         | DIVPD—Divide Packed Double-Precision Floating-Point Values                                   | 5-86  |
|         | DIVPS—Divide Packed Single-Precision Floating-Point Values                                   | 5-89  |
|         | DIVSD—Divide Scalar Double-Precision Floating-Point Value                                    | 5-92  |
|         | DIVSS—Divide Scalar Single-Precision Floating-Point Values                                   | 5-94  |
|         | VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory     | 5-96  |
|         | VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory     | 5-98  |
|         | CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values | 5-100 |

|  |       |
|--|-------|
| CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values                                   | 5-103 |
| CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers                                   | 5-106 |
| CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values                | 5-110 |
| VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers                                    | 5-114 |
| VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers                        | 5-116 |
| VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers                          | 5-118 |
| VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values   | 5-120 |
| VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value   | 5-123 |
| CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values                      | 5-127 |
| VCVTPS2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values                  | 5-130 |
| VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values                       | 5-132 |
| VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values                    | 5-134 |
| CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values                | 5-136 |
| VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values                                    | 5-139 |
| VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values                                    | 5-141 |
| CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer  | 5-143 |
| VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer                                 | 5-145 |
| CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value                  | 5-147 |
| CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value  | 5-149 |
| CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value  | 5-151 |
| CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value                  | 5-153 |
| CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer  | 5-155 |
| VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer                                 | 5-157 |
| CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers                  | 5-159 |
| VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers                   | 5-162 |
| VCVTTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers       | 5-164 |
| VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers         | 5-166 |
| CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values     | 5-168 |
| VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values | 5-171 |
| VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values      | 5-173 |
| VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values   | 5-175 |
| CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer                               | 5-177 |
| VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer                           | 5-179 |
| CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer                                      | 5-180 |
| VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer                           | 5-182 |
| VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values                        | 5-184 |
| VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values                        | 5-186 |
| VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values                          | 5-188 |
| VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values                          | 5-190 |
| VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value  | 5-192 |
| VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value  | 5-194 |
| VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes   | 5-196 |
| VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory  | 5-199 |
| VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory  | 5-201 |
| VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values                      | 5-203 |
| VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer                                    |       |

|  |       |
|--|-------|
| Values   | 5-209 |
| EXTRACTPS—Extract Packed Floating-Point Values   | 5-215 |
| VFIXUPIMMPD—Fix Up Special Packed Float64 Values   | 5-217 |
| VFIXUPIMMPS—Fix Up Special Packed Float32 Values   | 5-221 |
| VFIXUPIMMSD—Fix Up Special Scalar Float64 Value  | 5-225 |
| VFIXUPIMMSS—Fix Up Special Scalar Float32 Value  | 5-228 |
| VFMAADD132PD/VFMAADD213PD/VFMAADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values                               | 5-231 |
| VFMAADD132PS/VFMAADD213PS/VFMAADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values                               | 5-237 |
| VFMAADD132SD/VFMAADD213SD/VFMAADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values                               | 5-243 |
| VFMAADD132SS/VFMAADD213SS/VFMAADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values                               | 5-246 |
| VFMAADDSUB132PD/VFMAADDSUB213PD/VFMAADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values | 5-249 |
| VFMAADDSUB132PS/VFMAADDSUB213PS/VFMAADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values | 5-256 |
| VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values    | 5-263 |
| VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values    | 5-270 |
| VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values                             | 5-277 |
| VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values                             | 5-283 |
| VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values                             | 5-289 |
| VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values                             | 5-292 |
| VFNMAADD132PD/VFNMAADD213PD/VFNMAADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values                   | 5-295 |
| VFNMAADD132PS/VFNMAADD213PS/VFNMAADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values                   | 5-302 |
| VFNMAADD132SD/VFNMAADD213SD/VFNMAADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values                   | 5-308 |
| VFNMAADD132SS/VFNMAADD213SS/VFNMAADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values                   | 5-311 |
| VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values                 | 5-314 |
| VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values                 | 5-320 |
| VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values                 | 5-326 |
| VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values                 | 5-329 |
| VFPCLASSPD—Tests Types Of a Packed Float64 Values  | 5-332 |
| VFPCLASSPS—Tests Types Of a Packed Float32 Values  | 5-335 |
| VFPCLASSSD—Tests Types Of a Scalar Float64 Values  | 5-337 |
| VFPCLASSSS—Tests Types Of a Scalar Float32 Values  | 5-339 |
| VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices  | 5-341 |
| VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices  | 5-344 |
| VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword  | 5-347 |
| VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices  | 5-350 |
| VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values   | 5-353 |
| VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values   | 5-356 |
| VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value  | 5-360 |
| VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value  | 5-362 |
| VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector  | 5-364 |

|   |       |
|---|-------|
| VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector .....                       | 5-368 |
| VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar .....                              | 5-371 |
| VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector .....                        | 5-373 |
| VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values ..... | 5-375 |
| VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values .....        | 5-379 |
| INSERTPS—Insert Scalar Single-Precision Floating-Point Value .....  | 5-383 |
| MAXPD—Maximum of Packed Double-Precision Floating-Point Values .....                                      | 5-386 |
| MAXPS—Maximum of Packed Single-Precision Floating-Point Values .....                                      | 5-389 |
| MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value .....                                   | 5-392 |
| MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value .....                                   | 5-394 |
| MINPD—Minimum of Packed Double-Precision Floating-Point Values .....                                      | 5-396 |
| MINPS—Minimum of Packed Single-Precision Floating-Point Values .....                                      | 5-399 |
| MINSD—Return Minimum Scalar Double-Precision Floating-Point Value .....                                   | 5-402 |
| MINSS—Return Minimum Scalar Single-Precision Floating-Point Value .....                                   | 5-404 |
| MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values .....                                   | 5-406 |
| MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values .....                                   | 5-410 |
| MOVD/MOVQ—Move Doubleword and Quadword .....  | 5-414 |
| MOVQ—Move Quadword .....  | 5-417 |
| MOVDDUP—Replicate Double FP Values .....  | 5-420 |
| MOVDDQA, VMOVDDQA32/64—Move Aligned Packed Integer Values .....   | 5-423 |
| MOVDDQU, VMOVDDQU8/16/32/64—Move Unaligned Packed Integer Values .....                                    | 5-428 |
| MOVHLPD—Move Packed Single-Precision Floating-Point Values High to Low .....                              | 5-435 |
| MOVHPD—Move High Packed Double-Precision Floating-Point Value .....                                       | 5-437 |
| MOVHPS—Move High Packed Single-Precision Floating-Point Values .....                                      | 5-439 |
| MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High .....                              | 5-441 |
| MOVLPD—Move Low Packed Double-Precision Floating-Point Value .....  | 5-443 |
| MOVLPS—Move Low Packed Single-Precision Floating-Point Values .....                                       | 5-445 |
| MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint .....   | 5-447 |
| MOVNTDQ—Store Packed Integers Using Non-Temporal Hint .....   | 5-449 |
| MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint .....                 | 5-451 |
| MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint .....                 | 5-453 |
| MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value .....                                    | 5-455 |
| MOVSHDUP—Replicate Single FP Values .....   | 5-458 |
| MOVSLDUP—Replicate Single FP Values .....   | 5-461 |
| MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value .....                                    | 5-464 |
| MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values .....                                 | 5-467 |
| MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values .....                                 | 5-471 |
| PSADBW—Compute Sum of Absolute Differences .....  | 5-475 |
| MULPD—Multiply Packed Double-Precision Floating-Point Values .....  | 5-478 |
| MULPS—Multiply Packed Single-Precision Floating-Point Values .....  | 5-481 |
| MULSD—Multiply Scalar Double-Precision Floating-Point Value .....   | 5-484 |
| MULSS—Multiply Scalar Single-Precision Floating-Point Values .....  | 5-486 |
| ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values .....                            | 5-488 |
| ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values .....                            | 5-491 |
| PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value .....   | 5-494 |
| PACKSSWB/PACKSSDW—Pack with Signed Saturation .....   | 5-500 |
| PACKUSDW—Pack with Unsigned Saturation .....  | 5-508 |
| PACKUSWB—Pack with Unsigned Saturation .....  | 5-513 |
| PADDB/PADDW/PADDD/PADDQ—Add Packed Integers .....   | 5-518 |
| PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation .....                                     | 5-525 |
| PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation .....                               | 5-529 |
| PALIGNR—Byte Align .....  | 5-533 |
| PAND—Logical AND .....  | 5-536 |
| PANDN—Logical AND NOT .....   | 5-539 |
| PAVGB/PAVGW—Average Packed Integers .....   | 5-542 |
| VPBROADCASTM—Broadcast Mask to Vector Register .....  | 5-546 |
| PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ—Compare Packed Integers for Equality .....                                | 5-548 |
| PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ—Compare Packed Integers for Greater Than .....                            | 5-555 |

|  |       |
|--|-------|
| VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask .....  | 5-562 |
| VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask.....  | 5-565 |
| VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask .....   | 5-568 |
| VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask .....  | 5-571 |
| VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register .....                                     | 5-574 |
| VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register.....  | 5-576 |
| VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register .....                  | 5-578 |
| VPERMB—Permute Packed Bytes Elements .....   | 5-581 |
| VPERMD/VPERMW—Permute Packed Doublewords/Words Elements .....  | 5-583 |
| VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index .....   | 5-586 |
| VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index .....  | 5-588 |
| VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table.....  | 5-594 |
| See Exceptions Type E4NF.nb.....   | 5-595 |
| VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table.....   | 5-596 |
| VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values .....   | 5-602 |
| VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values .....  | 5-607 |
| VPERMPD—Permute Double-Precision Floating-Point Elements .....   | 5-612 |
| VPERMPS—Permute Single-Precision Floating-Point Elements.....  | 5-615 |
| VPERMQ—Qwords Element Permutation.....   | 5-618 |
| VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register .....                                      | 5-621 |
| VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register .....  | 5-623 |
| PEXTRB/PEXTRW/PEXTRD/PEXTRQ—Extract Integer .....  | 5-625 |
| VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values .....                                   | 5-628 |
| PMADDUBSW—Multiply and Add Packed Integers.....  | 5-631 |
| PMADDWD—Multiply and Add Packed Integers .....   | 5-633 |
| PINSRB/PINSRW/PINSRD/PINSRQ—Insert Integer.....  | 5-636 |
| VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword<br>Accumulators.....          | 5-640 |
| VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit<br>Accumulators' ..... | 5-642 |
| PMAXS/PMAXS/PMAXS/PMAXSQ—Maximum of Packed Signed Integers .....   | 5-644 |
| PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers .....  | 5-651 |
| PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers .....  | 5-655 |
| PMINSB/PMINSW—Minimum of Packed Signed Integers .....  | 5-659 |
| PMINS/PMINSQ—Minimum of Packed Signed Integers .....   | 5-663 |
| PMINUB/PMINUW—Minimum of Packed Unsigned Integers .....  | 5-667 |
| PMINUD/PMINUQ—Minimum of Packed Unsigned Integers.....   | 5-671 |
| VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register .....   | 5-675 |
| VPMOV2B2M/VPMOV2W2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask .....  | 5-678 |
| VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte .....  | 5-681 |
| VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word .....  | 5-685 |
| VPMOVQD/VPMOVSQD/VPMOVUSD—Down Convert QWord to DWord .....  | 5-689 |
| VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte.....   | 5-693 |
| VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word.....   | 5-697 |
| VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte.....  | 5-701 |
| PMOVSX—Packed Move with Sign Extend .....  | 5-705 |
| PMOVZX—Packed Move with Zero Extend .....  | 5-715 |
| PMULDQ—Multiply Packed Doubleword Integers.....  | 5-724 |
| PMULHRW—Multiply Packed Unsigned Integers with Round and Scale.....  | 5-727 |
| PMULHUW—Multiply Packed Unsigned Integers and Store High Result.....   | 5-730 |
| PMULHW—Multiply Packed Integers and Store High Result.....   | 5-733 |
| PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result.....   | 5-736 |
| PMULLW—Multiply Packed Integers and Store Low Result .....   | 5-740 |
| VPMULTISHIFTQB - Select Packed Unaligned Bytes from Quadword Sources .....   | 5-743 |
| PMULUDQ—Multiply Packed Unsigned Doubleword Integers.....  | 5-745 |
| POR—Bitwise Logical Or.....  | 5-748 |
| PROLD/PROLVD/PROLQ/PROLVQ—Bit Rotate Left .....  | 5-751 |
| PRORD/PRORVD/PRORQ/PRORVQ—Bit Rotate Right.....  | 5-755 |
| VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed                                 |       |



|   |       |
|---|-------|
| Dword, Signed Qword Indices .....   | 5-759 |
| PSHUFB—Packed Shuffle Bytes .....   | 5-763 |
| PSHUFHW—Shuffle Packed High Words .....   | 5-766 |
| PSHUFLW—Shuffle Packed Low Words .....  | 5-769 |
| PSHUFD—Shuffle Packed Doublewords .....   | 5-772 |
| PSLLDQ—Byte Shift Left .....  | 5-776 |
| PSLLW/PSLLD/PSLLQ—Bit Shift Left .....  | 5-778 |
| PSRAW/PSRAD/PSRAQ—Bit Shift Arithmetic Right .....  | 5-789 |
| PSRLDQ—Byte Shift Right .....   | 5-798 |
| PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical .....   | 5-800 |
| VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical .....   | 5-811 |
| VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical .....  | 5-816 |
| PSUBB/PSUBW/PSUBD/PSUBQ—Packed Integer Subtract .....   | 5-821 |
| PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation .....  | 5-829 |
| PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation .....  | 5-833 |
| VPTESTNMB/W/D/Q—Logical NAND and Set .....  | 5-837 |
| PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ—Unpack High Data .....   | 5-840 |
| PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data .....  | 5-849 |
| SHUFF32x4/SHUFF64x2/SHUFI32x4/SHUFI64x2—Shuffle Packed Values at 128-bit Granularity .....  | 5-859 |
| SHUFPD—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values .....   | 5-864 |
| SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values .....                                   | 5-869 |
| SQRTPD—Square Root of Double-Precision Floating-Point Values .....  | 5-873 |
| SQRTPS—Square Root of Single-Precision Floating-Point Values .....  | 5-876 |
| SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value .....  | 5-879 |
| SQRTSS—Compute Square Root of Scalar Single-Precision Value .....   | 5-881 |
| VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic .....   | 5-883 |
| VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask .....  | 5-886 |
| VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic .....   | 5-889 |
| PXOR/PXORD/PXORQ—Exclusive Or .....   | 5-894 |
| VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values .....   | 5-897 |
| VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values .....   | 5-901 |
| VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values .....   | 5-904 |
| VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values .....   | 5-907 |
| VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values .....   | 5-910 |
| VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value .....   | 5-912 |
| VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values .....   | 5-914 |
| VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value .....   | 5-916 |
| VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values .....   | 5-918 |
| VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value .....  | 5-921 |
| VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values .....   | 5-923 |
| VREDUCES—Perform a Reduction Transformation on a Scalar Float32 Value .....   | 5-925 |
| VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits .....  | 5-927 |
| VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits .....   | 5-930 |
| VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits .....  | 5-932 |
| VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits .....   | 5-935 |
| VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values .....   | 5-937 |
| VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value .....  | 5-939 |
| VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values .....   | 5-941 |
| VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value .....  | 5-943 |
| VSCALEFPD—Scale Packed Float64 Values With Float64 Values .....   | 5-945 |
| VSCALEFSD—Scale Scalar Float64 Values With Float64 Values .....   | 5-948 |
| VSCALEFPS—Scale Packed Float32 Values With Float32 Values .....   | 5-950 |
| VSCALEFSS—Scale Scalar Float32 Value With Float32 Value .....   | 5-952 |
| VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed<br>Dword and Qword Indices ..... | 5-954 |
| SUBPD—Subtract Packed Double-Precision Floating-Point Values .....  | 5-958 |
| SUBPS—Subtract Packed Single-Precision Floating-Point Values .....  | 5-961 |
| SUBSD—Subtract Scalar Double-Precision Floating-Point Value .....   | 5-964 |
| SUBSS—Subtract Scalar Single-Precision Floating-Point Value .....   | 5-966 |

|  |       |
|--|-------|
| UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS ..... | 5-968 |
| UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS ..... | 5-970 |
| UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values .....      | 5-972 |
| UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values .....      | 5-976 |
| UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values .....       | 5-980 |
| UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values .....       | 5-984 |
| XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values .....             | 5-988 |
| XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values .....             | 5-991 |

## CHAPTER 6 INSTRUCTION SET REFERENCE - OPMASK

|   |      |
|---|------|
| 6.1 MASK INSTRUCTIONS .....   | 6-1  |
| KADDW/KADDB/KADDQ/KADDD—ADD Two Masks .....                           | 6-1  |
| KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks .....               | 6-3  |
| KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks .....       | 6-4  |
| KMOVW/KMOVB/KMOVQ/KMOVD—Move from and to Mask Registers .....         | 6-5  |
| KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers .....            | 6-7  |
| KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register .....                       | 6-8  |
| KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks .....                    | 6-9  |
| KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks And Set Flags .....      | 6-10 |
| KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers .....   | 6-12 |
| KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers .....  | 6-14 |
| KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks .....          | 6-16 |
| KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags ..... | 6-17 |
| KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks .....               | 6-19 |

## CHAPTER 7 ADDITIONAL 512-BIT INSTRUCTION EXTENSIONS

|   |      |
|---|------|
| 7.1 Detection of 512-bit Instruction Extensions .....   | 7-1  |
| 7.2 Instruction SET Reference .....   | 7-2  |
| VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error .....   | 7-3  |
| VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error .....   | 7-5  |
| VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error .....   | 7-7  |
| VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error .....  | 7-9  |
| VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error .....   | 7-11 |
| VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error .....  | 7-13 |
| VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error .....                                 | 7-15 |
| VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error .....                                  | 7-17 |
| VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error .....                                 | 7-19 |
| VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error .....                                  | 7-21 |
| VGATHERPFODPS/VGATHERPFQOPS/VGATHERPFODPD/VGATHERPFQOPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using TO Hint .....                          | 7-23 |
| VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint .....                          | 7-25 |
| VSCATTERPFODPS/VSCATTERPFQOPS/VSCATTERPFODPD/VSCATTERPFQOPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using TO Hint with Intent to Write ..... | 7-27 |
| VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write ..... | 7-29 |

## CHAPTER 8 INTEL® SHA EXTENSIONS

|       |  |      |
|-------|--|------|
| 8.1   | Overview   | 8-1  |
| 8.2   | Detection of Intel SHA Extensions  | 8-1  |
| 8.2.1 | Common Transformations and Primitive Functions   | 8-1  |
| 8.3   | SHA Extensions Reference   | 8-2  |
|       | SHA1RNDS4—Perform Four Rounds of SHA1 Operation  | 8-3  |
|       | SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds                            | 8-5  |
|       | SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords     | 8-6  |
|       | SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords             | 8-7  |
|       | SHA256RNDS2—Perform Two Rounds of SHA256 Operation                                     | 8-8  |
|       | SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords | 8-10 |
|       | SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords         | 8-11 |

## CHAPTER 9 ADDITIONAL NEW INSTRUCTIONS

|     |   |     |
|-----|---|-----|
| 9.1 | Instruction Format  | 9-1 |
| 9.2 | INSTRUCTION SET REFERENCE   | 9-1 |
|     | PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint | 9-2 |

## CHAPTER 10 MEMORY INSTRUCTIONS

|        |   |       |
|--------|---|-------|
| 10.1   | Detection of New Instructions                                       | 10-1  |
| 10.2   | Persistent Memory   | 10-1  |
| 10.2.1 | Accessing Persistent Memory   | 10-1  |
| 10.2.2 | Managing Persistence  | 10-1  |
| 10.3   | Instruction Format  | 10-2  |
|        | CLFLUSHOPT — Flush a Cache Line (THIS IS AN EXAMPLE)                | 10-3  |
| 10.4   | INSTRUCTION SET REFERENCE   | 10-3  |
|        | CLFLUSHOPT—Flush a Cache Line Optimized                             | 10-4  |
|        | CLWB—Cache Line Write Back  | 10-6  |
|        | PCOMMIT—Persistent Commit   | 10-8  |
| 10.5   | Persistent Memory Configuration and Enumeration of Platform Support | 10-10 |
| 10.6   | PCOMMIT — Virtualization Support                                    | 10-10 |
| 10.7   | PCOMMIT and SGX Interaction   | 10-10 |



# TABLES

|      | PAGE  |      |
|------|---|------|
| 2-1  | 512-bit Instruction Groups in the Intel AVX-512 Family . . . . .                                    | 2-2  |
| 2-2  | Feature flag Collection Required of 256/128 Bit Vector Lengths for Each Instruction Group . . . . . | 2-4  |
| 2-3  | Instruction Mnemonics That Do Not Support EVEX.128 Encoding . . . . .                               | 2-4  |
| 2-4  | Characteristics of Three Rounding Control Interfaces . . . . .                                      | 2-8  |
| 2-5  | Static Rounding Mode . . . . .  | 2-9  |
| 2-6  | SIMD Instructions Requiring Explicitly Aligned Memory . . . . .                                     | 2-10 |
| 2-7  | Instructions Not Requiring Explicit Memory Alignment . . . . .                                      | 2-11 |
| 2-8  | Information Returned by CPUID Instruction . . . . .   | 2-13 |
| 2-9  | Highest CPUID Source Operand for Intel 64 and IA-32 Processors . . . . .                            | 2-21 |
| 2-10 | Processor Type Field . . . . .  | 2-22 |
| 2-11 | Feature Information Returned in the ECX Register . . . . .  | 2-24 |
| 2-12 | More on Feature Information Returned in the EDX Register . . . . .                                  | 2-26 |
| 2-13 | Encoding of Cache and TLB Descriptors . . . . .   | 2-28 |
| 2-14 | Structured Extended Feature Leaf, Function 0, EBX Register . . . . .                                | 2-31 |
| 2-15 | Processor Brand String Returned with Pentium 4 Processor . . . . .                                  | 2-33 |
| 2-16 | Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings . . . . .                  | 2-35 |
| 3-1  | XCRO Processor State Components . . . . .   | 3-2  |
| 3-2  | CR4 Bits for AVX-512 Foundation Instructions Technology Support . . . . .                           | 3-3  |
| 3-3  | Layout of XSAVE Area For Processor Supporting YMM State . . . . .                                   | 3-4  |
| 3-4  | XSAVE Header Format . . . . .   | 3-4  |
| 3-5  | XSAVE Save Area Layout for YMM_Hi128 State (Ext_Save_Area_2) . . . . .                              | 3-4  |
| 3-6  | XSAVE Save Area Layout for Opmask Registers . . . . .   | 3-5  |
| 3-7  | XSAVE Save Area Layout for ZMM State of the High 256 Bits of ZMM0-ZMM15 Registers . . . . .         | 3-5  |
| 3-8  | XSAVE Save Area Layout for ZMM State of ZMM16-ZMM31 Registers . . . . .                             | 3-5  |
| 3-9  | XRSTOR Action on MXCSR, XMM Registers, YMM Registers . . . . .                                      | 3-6  |
| 3-10 | XSAVE Action on MXCSR, XMM, YMM Register . . . . .  | 3-6  |
| 3-11 | Processor Supplied Init Values XRSTOR May Use . . . . .   | 3-7  |
| 4-1  | EVEX Prefix Bit Field Functional Grouping . . . . .   | 4-2  |
| 4-2  | 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits . . . . .                      | 4-3  |
| 4-3  | EVEX Encoding Register Specifiers in 32-bit Mode . . . . .  | 4-3  |
| 4-4  | Opmask Register Specifier Encoding . . . . .  | 4-4  |
| 4-5  | Compressed Displacement (DISP8*N) Affected by Embedded Broadcast . . . . .                          | 4-5  |
| 4-6  | EVEX DISP8*N For Instructions Not Affected by Embedded Broadcast . . . . .                          | 4-5  |
| 4-7  | EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions . . . . .             | 4-7  |
| 4-8  | OS XSAVE Enabling Requirements of Instruction Categories . . . . .                                  | 4-7  |
| 4-9  | Opcode Independent, State Dependent EVEX Bit Fields . . . . .                                       | 4-8  |
| 4-10 | #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields . . . . .                                 | 4-8  |
| 4-11 | #UD Conditions of Opmask Related Encoding Field . . . . .   | 4-9  |
| 4-12 | #UD Conditions Dependent on EVEX.b Context . . . . .  | 4-9  |
| 4-13 | EVEX-Encoded Instruction Exception Class Summary . . . . .  | 4-10 |
| 4-14 | EVEX Instructions in each Exception Class . . . . .   | 4-10 |
| 4-15 | Type E1 Class Exception Conditions . . . . .  | 4-13 |
| 4-16 | Type E1NF Class Exception Conditions . . . . .  | 4-14 |
| 4-17 | Type E2 Class Exception Conditions . . . . .  | 4-15 |
| 4-18 | Type E3 Class Exception Conditions . . . . .  | 4-16 |
| 4-19 | Type E3NF Class Exception Conditions . . . . .  | 4-17 |
| 4-20 | Type E4 Class Exception Conditions . . . . .  | 4-18 |
| 4-21 | Type E4NF Class Exception Conditions . . . . .  | 4-19 |
| 4-22 | Type E5 Class Exception Conditions . . . . .  | 4-20 |
| 4-23 | Type E5NF Class Exception Conditions . . . . .  | 4-21 |
| 4-24 | Type E6 Class Exception Conditions . . . . .  | 4-22 |
| 4-25 | Type E6NF Class Exception Conditions . . . . .  | 4-23 |
| 4-26 | Type E7NM Class Exception Conditions . . . . .  | 4-24 |
| 4-27 | Type E9 Class Exception Conditions . . . . .  | 4-25 |
| 4-28 | Type E9NF Class Exception Conditions . . . . .  | 4-26 |

|      |   |       |
|------|---|-------|
| 4-29 | Type E10 Class Exception Conditions   | 4-27  |
| 4-30 | Type E10NF Class Exception Conditions   | 4-28  |
| 4-31 | Type E11 Class Exception Conditions   | 4-29  |
| 4-32 | Type E12 Class Exception Conditions   | 4-30  |
| 4-33 | Type E12NP Class Exception Conditions   | 4-31  |
| 4-34 | TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)                                | 4-32  |
| 4-35 | TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)                             | 4-33  |
| 5-1  | Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations  | 5-8   |
| 5-2  | Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations  | 5-9   |
| 5-3  | Comparison Predicate for CMPPD and CMPPS Instructions   | 5-61  |
| 5-4  | Pseudo-Op and CMPPD Implementation  | 5-62  |
| 5-5  | Pseudo-Op and VCMPPD Implementation   | 5-63  |
| 5-6  | Pseudo-Op and CMPPS Implementation  | 5-68  |
| 5-7  | Pseudo-Op and VCMPPS Implementation   | 5-69  |
| 5-8  | Pseudo-Op and CMPSD Implementation  | 5-74  |
| 5-9  | Pseudo-Op and VCMPSD Implementation   | 5-74  |
| 5-10 | Pseudo-Op and CMPSS Implementation  | 5-78  |
| 5-11 | Pseudo-Op and VCMPS Implementation  | 5-78  |
| 5-12 | Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions                                     | 5-124 |
| 5-13 | Classifier Operations for VFPCCLASSPD/SD/PS/SS  | 5-332 |
| 5-14 | VGETEXPPD/SD Special Cases  | 5-353 |
| 5-15 | VGETEXPPS/SS Special Cases  | 5-356 |
| 5-16 | GetMant() Special Float Values Behavior   | 5-365 |
| 5-17 | Pseudo-Op and VPCMP* Implementation   | 5-563 |
| 5-18 | Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values   | 5-884 |
| 5-19 | Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]                     | 5-898 |
| 5-20 | Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS and MAX, MAX_ABS                            | 5-898 |
| 5-21 | Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, ( $ a  =  b $ , $a > 0$ , $b < 0$ ) | 5-898 |
| 5-22 | VRCP14PD/VRCP14SD Special Cases   | 5-910 |
| 5-23 | VRCP14PS/VRCP14SS Special Cases   | 5-914 |
| 5-24 | VREDUCEPD/SD/PS/SS Special Cases  | 5-919 |
| 5-25 | VRNDSCALEPD/SD/PS/SS Special Cases  | 5-928 |
| 5-26 | VRSQRT14PD Special Cases  | 5-938 |
| 5-27 | VRSQRT14SD Special Cases  | 5-940 |
| 5-28 | VRSQRT14PS Special Cases  | 5-942 |
| 5-29 | VRSQRT14SS Special Cases  | 5-944 |
| 5-30 | VSCALEFPD/SD/PS/SS Special Cases  | 5-945 |
| 5-31 | Additional VSCALEFPD/SD Special Cases   | 5-946 |
| 5-32 | Additional VSCALEFPS/SS Special Cases   | 5-950 |
| 7-1  | Special Values Behavior   | 7-4   |
| 7-2  | Special Values Behavior   | 7-6   |
| 7-3  | VRCP28PD Special Cases  | 7-8   |
| 7-4  | VRCP28SD Special Cases  | 7-10  |
| 7-5  | VRCP28PS Special Cases  | 7-12  |
| 7-6  | VRCP28SS Special Cases  | 7-14  |
| 7-7  | VRSQRT28PD Special Cases  | 7-16  |
| 7-8  | VRSQRT28SD Special Cases  | 7-18  |
| 7-9  | VRSQRT28PS Special Cases  | 7-20  |
| 7-10 | VRSQRT28SS Special Cases  | 7-22  |

# FIGURES

PAGE

|              |  |       |
|--------------|--|-------|
| Figure 1-1.  | 512-Bit Wide Vectors and SIMD Register Set .....   | 1-2   |
| Figure 2-1.  | Procedural Flow of Application Detection of AVX-512 Foundation Instructions .....                              | 2-1   |
| Figure 2-2.  | Procedural Flow of Application Detection of 512-bit Instruction Groups .....                                   | 2-2   |
| Figure 2-3.  | Procedural Flow of Application Detection of Intel AVX-512 Instructions Operating at Vector Lengths < 512 ..... | 2-3   |
| Figure 2-4.  | Version Information Returned by CPUID in EAX .....   | 2-22  |
| Figure 2-5.  | Feature Information Returned in the ECX Register .....   | 2-24  |
| Figure 2-6.  | Feature Information Returned in the EDX Register .....   | 2-26  |
| Figure 2-7.  | Determination of Support for the Processor Brand String .....  | 2-33  |
| Figure 2-8.  | Algorithm for Extracting Maximum Processor Frequency .....   | 2-34  |
| Figure 3-1.  | Bit Vector and XCRO Layout of Extended Processor State Components .....  | 3-2   |
| Figure 4-1.  | AVX-512 Instruction Format and the EVEX Prefix .....   | 4-1   |
| Figure 4-2.  | Bit Field Layout of the EVEX Prefix .....  | 4-2   |
| Figure 5-1.  | VBROADCASTSS Operation (VEX.256 encoded version) .....   | 5-43  |
| Figure 5-2.  | VBROADCASTSS Operation (VEX.128-bit version) .....   | 5-43  |
| Figure 5-3.  | VBROADCASTSD Operation (VEX.256-bit version) .....   | 5-43  |
| Figure 5-4.  | VBROADCASTF128 Operation (VEX.256-bit version) .....   | 5-43  |
| Figure 5-5.  | VBROADCASTF64X4 Operation (512-bit version with writemask all 1s) .....  | 5-44  |
| Figure 5-6.  | VPBROADCASTD Operation (VEX.256 encoded version) .....   | 5-53  |
| Figure 5-7.  | VPBROADCASTD Operation (128-bit version) .....   | 5-53  |
| Figure 5-8.  | VPBROADCASTQ Operation (256-bit version) .....   | 5-54  |
| Figure 5-9.  | VBROADCASTI128 Operation (256-bit version) .....   | 5-54  |
| Figure 5-10. | VBROADCASTI256 Operation (512-bit version) .....   | 5-54  |
| Figure 5-11. | CVTDQ2PD (VEX.256 encoded version) .....   | 5-101 |
| Figure 5-12. | VCVTPD2DQ (VEX.256 encoded version) .....  | 5-107 |
| Figure 5-13. | VCVTPD2PS (VEX.256 encoded version) .....  | 5-111 |
| Figure 5-14. | VCVTPH2PS (128-bit Version) .....  | 5-121 |
| Figure 5-15. | VCVTPS2PH (128-bit Version) .....  | 5-123 |
| Figure 5-16. | CVTPS2PD (VEX.256 encoded version) .....   | 5-137 |
| Figure 5-17. | VCVTTPD2DQ (VEX.256 encoded version) .....   | 5-160 |
| Figure 5-18. | 64-bit Super Block of SAD Operation in VDBPSADBW .....   | 5-197 |
| Figure 5-19. | VFIXUPIMMPD Immediate Control Description .....  | 5-219 |
| Figure 5-20. | VFIXUPIMMPS Immediate Control Description .....  | 5-223 |
| Figure 5-21. | VFIXUPIMMSD Immediate Control Description .....  | 5-227 |
| Figure 5-22. | VFIXUPIMMSS Immediate Control Description .....  | 5-230 |
| Figure 5-23. | Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS .....                                    | 5-332 |
| Figure 5-24. | VGETEXPPS Functionality On Normal Input values .....   | 5-357 |
| Figure 5-25. | Imm8 Controls for VGETMANTPD/SD/PS/SS .....  | 5-364 |
| Figure 5-26. | VMOVDDUP Operation .....   | 5-421 |
| Figure 5-27. | MOVSHDUP Operation .....   | 5-459 |
| Figure 5-28. | MOVSLDUP Operation .....   | 5-461 |
| Figure 5-29. | 256-bit VPALIGN Instruction Operation .....  | 5-534 |
| Figure 5-30. | VPERMILPD Operation .....  | 5-603 |
| Figure 5-31. | VPERMILPD Shuffle Control .....  | 5-603 |
| Figure 5-32. | VPERMILPS Operation .....  | 5-608 |
| Figure 5-33. | VPERMILPS Shuffle Control .....  | 5-608 |
| Figure 5-34. | 256-bit VPSHUFD Instruction Operation .....  | 5-773 |
| Figure 5-35. | 256-bit VPUNPCKHDQ Instruction Operation .....   | 5-842 |
| Figure 5-36. | 128-bit PUNPCKLBW Instruction Operation using 64-bit Operands .....  | 5-851 |
| Figure 5-37. | 256-bit VPUNPCKLDQ Instruction Operation .....   | 5-851 |
| Figure 5-38. | 256-bit VSHUFPD Operation of Four Pairs of DP FP Values .....  | 5-865 |
| Figure 5-39. | 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result .....            | 5-870 |
| Figure 5-40. | Imm8 Controls for VRANGEPD/SD/PS/SS .....  | 5-897 |
| Figure 5-41. | Imm8 Controls for VREDUCEPD/SD/PS/SS .....   | 5-918 |
| Figure 5-42. | Imm8 Controls for VRNDSCALEPD/SD/PS/SS .....   | 5-928 |
| Figure 5-43. | VUNPCKHPS Operation .....  | 5-977 |

|              |   |       |
|--------------|---|-------|
| Figure 5-44. | VUNPCKLPS Operation.....  | 5-985 |
| Figure 7-1.  | Procedural Flow of Application Detection of 512-bit Instructions..... | 7-1   |



# CHAPTER 1

## FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS

---

### 1.1 ABOUT THIS DOCUMENT

This document describes the software programming interfaces of Intel® architecture instruction extensions for future Intel processor generations. The instruction set extensions cover a diverse range of application domains and programming usages. There are 512-bit SIMD vector instruction extensions, instruction set extensions targeting memory protection issues such as buffer overruns, and extensions targeting secure hash algorithm (SHA) accelerations like SHA1 and SHA256.

The 512-bit SIMD vector SIMD extensions, referred to as Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions, deliver comprehensive set of functionality and higher performance than AVX and AVX2 family of instructions. AVX and AVX2 are covered in *Intel® 64 and IA-32 Architectures Software Developer's Manual sets*. The reader can refer to them for basic and more background information related to various features referenced in this document.

The base of the 512-bit SIMD instruction extensions are referred to as Intel® Intel AVX-512 Foundation instructions. They include extensions of the AVX and AVX2 family of SIMD instructions but are encoded using a new encoding scheme with support for 512-bit vector registers, up to 32 vector registers in 64-bit mode, and conditional processing using opmask registers.

Chapters 2 through 6 are devoted to the programming interfaces of the Intel AVX-512 Foundation instruction set, additional 512-bit instruction extensions in the Intel AVX-512 family targeting broad application domains, and instruction set extensions encoded using the EVEX prefix encoding scheme to operate at vector lengths smaller than 512-bits.

Chapter 7 covers additional 512-bit SIMD instruction extensions that targets specific application domain, Intel AVX-512 Exponential and Reciprocal instructions for certain transcendental mathematical computations, and Intel AVX-512 Prefetch instructions for specific prefetch operations.

Chapter 8 covers instruction set extensions targeted for SHA acceleration. Chapter 9 describes instruction set extensions that offer software tools with capability to address memory protection issues such as buffer overruns. For an overview and detailed descriptions of hardware -accelerated SHA extensions, and Intel® Memory Protection Extensions (Intel® MPX), see the respective chapters.

Chapter 10 covers instructions operating on general purpose registers in future Intel processors. Chapter 11 describes the architecture of Intel® Processor Trace, which allows software to capture data packets with low overhead and to reconstruct detailed control flow information of program execution.

### 1.2 INTEL® AVX-512 INSTRUCTIONS ARCHITECTURE OVERVIEW

Intel Intel AVX-512 Foundation instructions are a natural extension to AVX and AVX2. It introduces the following architectural enhancements:

- Support for 512-bit wide vectors and SIMD register set. 512-bit register state is managed by the operating system using XSAVE/XRSTOR instructions introduced in 45 nm Intel 64 processors (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
- Support for 16 new, 512-bit SIMD registers (for a total of 32 SIMD registers, ZMM0 through ZMM31) in 64-bit mode. The extra 16 registers state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT.
- Support for 8 new opmask registers (k0 through k7) used for conditional execution and efficient merging of destination operands. Again, the opmask register state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT instructions
- A new encoding prefix (referred to as EVEX) to support additional vector length encoding up to 512 bits. The EVEX prefix builds upon the foundations of VEX prefix, to provide compact, efficient encoding for functionality available to VEX encoding plus the following enhanced vector capabilities:

- opmasks
- embedded broadcast
- instruction prefix-embedded rounding control
- compressed address displacements

### 1.2.1 512-Bit Wide SIMD Register Support

AVX-512 instructions support 512-bit wide SIMD registers (ZMM0-ZMM31). The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers and the lower 128-bit are aliased to the respective 128-bit XMM registers.

### 1.2.2 32 SIMD Register Support

AVX-512 instructions also support for 32 SIMD registers in 64-bit mode (XMM0-XMM31, YMM0-YMM31 and ZMM0-ZMM31). The number of available vector registers in 32-bit mode is still 8.

### 1.2.3 Eight Opmask Register Support

AVX-512 instructions support 8 opmask registers (k0-k7). The width of each opmask register is architecturally defined of size MAX\_KL (64 bits). Seven of the eight opmask registers (k1-k7) can be used in conjunction with EVEX-encoded Intel AVX-512 Foundation instructions to provide conditional execution and efficient merging of data elements in the destination operand. The encoding of opmask register k0 is typically used when all data elements (unconditional processing) are desired. Additionally, the opmask registers are also used as vector flags/element-level vector sources to introduce novel SIMD functionality as seen in new instructions such as VCOM-PRESSPS.

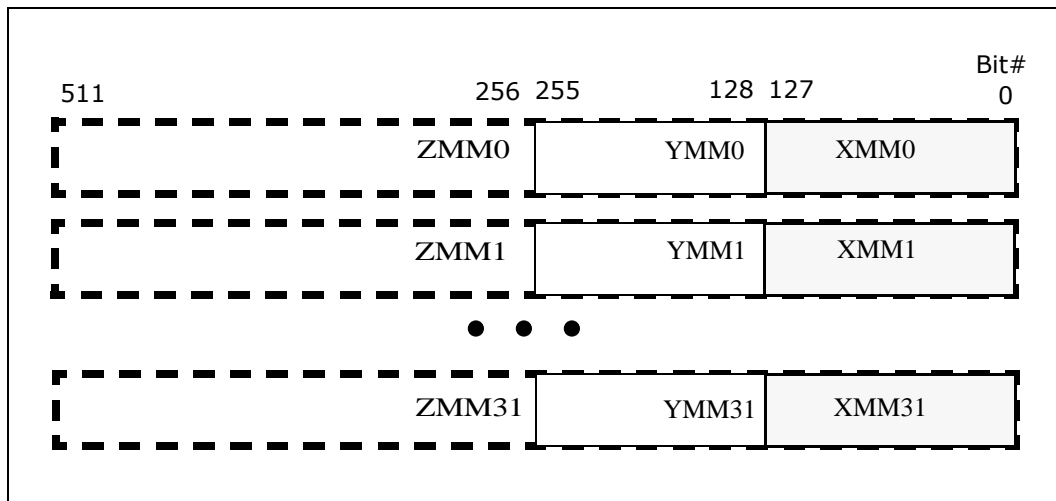


Figure 1-1. 512-Bit Wide Vectors and SIMD Register Set

### 1.2.4 Instruction Syntax Enhancement

The architecture of EVEX encoding enhances vector instruction encoding scheme in the following way:

- 512-bit vector-length, up to 32 ZMM registers, and enhanced vector programming environment are supported using the enhanced VEX (EVEX).

The EVEX prefix provides more encodable bit fields than VEX prefix. In addition to encoding 32 ZMM registers in 64-bit mode, instruction encoding using the EVEX can directly encode 7 (out of 8) opmask register operands to provide

conditional processing in vector instruction programming. The enhanced vector programming environment can be explicitly expressed in the instruction syntax to include the following elements:

- An opmask operand: the opmask registers are expressed using the notation “k1” through “k7”. An EVEX-encoded instruction supporting conditional vector operation using the opmask register k1 is expressed by attaching the notation {k1} next to the destination operand. The use of this feature is optional for most instructions. There are two types of masking (merging and zeroing) differentiated using the EVEX.z bit ({z} in instruction signature).
- Embedded broadcast may be supported for some instructions on the source operand that can be encoded as a memory vector. Data elements of a memory vector may be conditionally fetched or written to.
- For instruction syntax that operates only on floating-point data in SIMD registers with rounding semantics, the EVEX can provide explicit rounding control within the EVEX bit fields at either scalar or 512-bit vector length.

In AVX-512 instructions, vector addition of all elements of the source operands can be expressed in the same syntax as AVX instruction:

```
VADDPS zmm1, zmm2, zmm3
```

Additionally, the EVEX encoding scheme of Intel AVX-512 Foundation can express conditional vector addition as

```
VADDPS zmm1 {k1}{z}, zmm2, zmm3
```

where

- conditional processing and updates to destination is expressed with an opmask register,
- zeroing behavior of the opmask selected destination element is expressed by the {z} modifier (with merging as the default if no modifier specified),

Note that some SIMD instructions supporting three-operand syntax but processing only less or equal than 128-bits of data are considered part of the 512-bit SIMD instruction set extensions, because bits MAX\_VL-1:128 of the destination register are zeroed by the processor. The same rule applies to instructions operating on 256-bits of data where bits MAX\_VL-1:256 of the destination register are zeroed.

## 1.2.5 EVEX Instruction Encoding Support

Intel AVX-512 instructions employ a new encoding prefix, referred to as EVEX, in the Intel 64 and IA-32 instruction encoding format. Instruction encoding using the EVEX prefix provides the following capabilities:

- Direct encoding of a SIMD register operand within EVEX (similar to VEX). This provides instruction syntax support for three source operands.
- Compaction of REX prefix functionality and extended SIMD register encoding: The equivalent REX-prefix compaction functionality offered by the VEX prefix is provided within EVEX. Furthermore, EVEX extends the operand encoding capability to allow direct addressing of up to 32 ZMM registers in 64-bit mode.
- Compaction of SIMD prefix functionality and escape byte encoding: The functionality of SIMD prefix (66H, F2H, F3H) on opcode is equivalent to an opcode extension field to introduce new processing primitives. This functionality is provided in the VEX prefix encoding scheme and employed within the EVEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the EVEX prefix encoding.
- Most EVEX-encoded SIMD numeric and data processing instruction semantics with memory operand have relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 2.6, “Memory Alignment”).
- Direct encoding of a opmask operand within the EVEX prefix. This provides instruction syntax support for conditional vector-element operation and merging of destination operand using an opmask register (k1-k7).
- Direct encoding of a broadcast attribute for instructions with a memory operand source. This provides instruction syntax support for elements broadcasting of the second operand before being used in the actual operation.

- Compressed memory address displacements for a more compact instruction encoding byte sequence.
- EVEX encoding applies to SIMD instructions operating on XMM, YMM and ZMM registers. EVEX is not supported for instructions operating on MMX or x87 registers. Details of EVEX instruction encoding are discussed in Chapter 4.

## CHAPTER 2

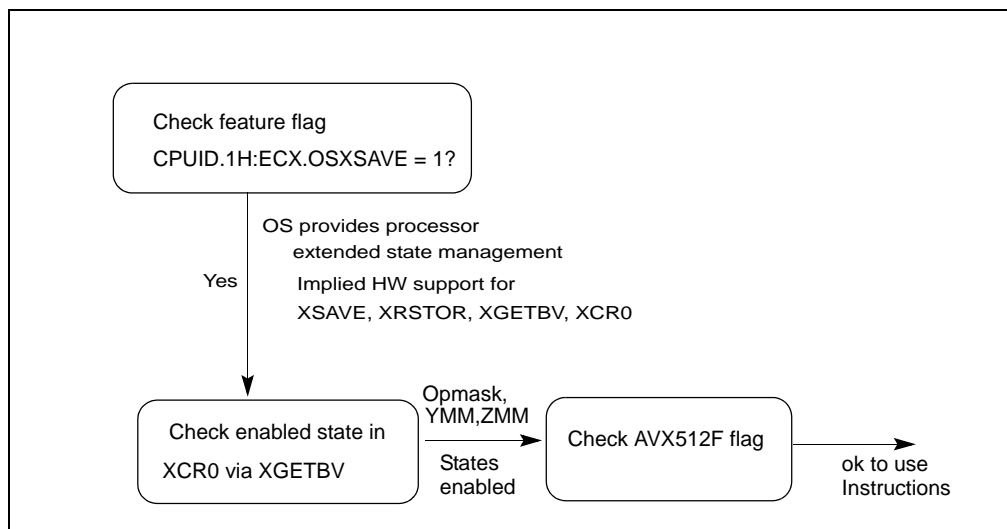
# INTEL® AVX-512 APPLICATION PROGRAMMING MODEL

The application programming model for AVX-512 Foundation instructions, several member groups of the Intel AVX-512 family (described in Chapter 5) and other 512-bit instructions (described in Chapter 7) extend from that of AVX and AVX2 with differences detailed in this chapter.

## 2.1 DETECTION OF AVX-512 FOUNDATION INSTRUCTIONS

The majority of AVX-512 Foundation instructions are encoded using the EVEX encoding scheme. EVEX-encoded instructions can operate on the 512-bit ZMM register state plus 8 opmask registers. The opmask instructions in AVX-512 Foundation instructions operate only on opmask registers or with a general purpose register. System software requirements to support ZMM state and opmask instructions are described in Chapter 3, “System Programming For Intel® AVX-512”.

Processor support of AVX-512 Foundation instructions is indicated by CPUID.(EAX=07H, ECX=0):EBX.AVX512F[bit 16] = 1. Detection of AVX-512 Foundation instructions operating on ZMM states and opmask registers need to follow the general procedural flow in Figure 2-1.



**Figure 2-1. Procedural Flow of Application Detection of AVX-512 Foundation Instructions**

Prior to using AVX-512 Foundation instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor’s support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

- 1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use<sup>1</sup>)
- 2) Execute XGETBV and verify that XCR0[7:5] = ‘111b’ (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = ‘11b’ (XMM state and YMM state are enabled by OS).
- 3) Detect CPUID.0x7.0:EBX.AVX512F[bit 16] = 1.

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XCR0 register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.

## 2.2 DETECTION OF 512-BIT INSTRUCTION GROUPS OF INTEL® AVX-512 FAMILY

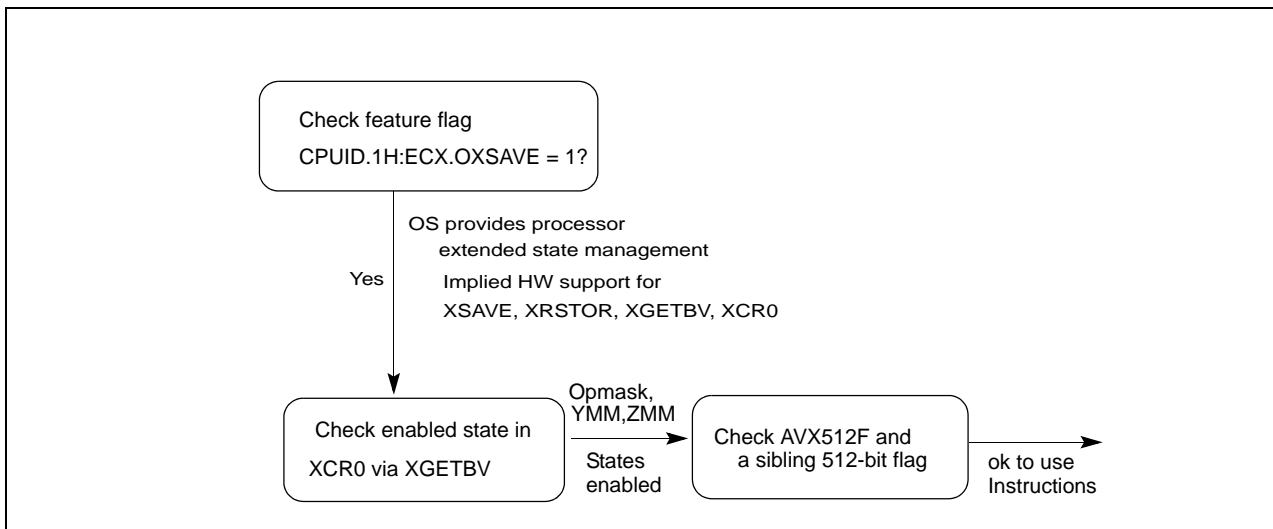
In addition to the Intel AVX-512 Foundation instructions, Intel AVX-512 family provides several additional 512-bit extensions in groups of instructions, each group is enumerated by a CPUID leaf 7 feature flag and can be encoded via EVEX.L'L field to support operation at vector lengths smaller than 512 bits. These instruction groups are listed in Table 2-1.

**Table 2-1. 512-bit Instruction Groups in the Intel AVX-512 Family**

| CPUID Leaf 7 Feature Flag Bit      | Feature Flag abbreviation of 512-bit Instruction Group | SW Detection Flow |
|------------------------------------|--|-------------------|
| CPUID.(EAX=07H, ECX=0):EBX[bit 16] | AVX512F (AVX-512 Foundation)                           | Figure 2-1        |
| CPUID.(EAX=07H, ECX=0):EBX[bit 28] | AVX512CD   | Figure 2-2        |
| CPUID.(EAX=07H, ECX=0):EBX[bit 17] | AVX512DQ   | Figure 2-2        |
| CPUID.(EAX=07H, ECX=0):EBX[bit 30] | AVX512BW   | Figure 2-2        |
| CPUID.(EAX=07H, ECX=0):EBX[bit 21] | AVX512IFMA   | Figure 2-2        |
| CPUID.(EAX=07H, ECX=0):ECX[bit 01] | AVX512VBMI   | Figure 2-2        |

Software must follow the detection procedure for the 512-bit AVX-512 Foundation instructions as described in Section 2.1.

Detection of other 512-bit sibling instruction groups listed in Table 2-1 (excluding AVX512F) follows the procedure described in Figure 2-2:



**Figure 2-2. Procedural Flow of Application Detection of 512-bit Instruction Groups**

To illustrate the detection procedure for 512-bit instructions enumerated by AVX512CD, the following sequence is strongly recommended.

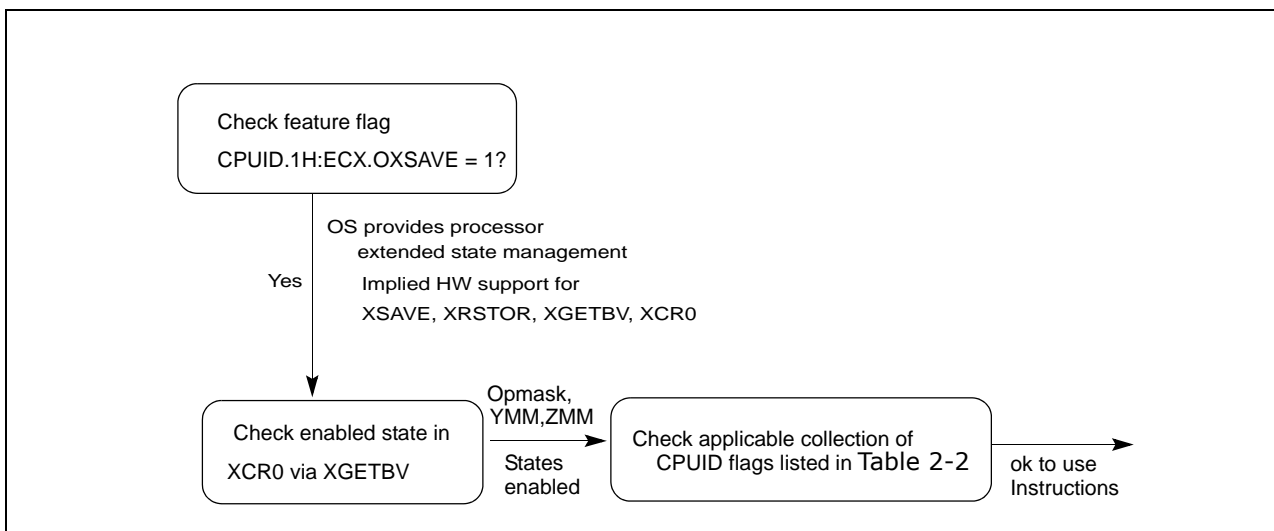
- 1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use)
- 2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
- 3) Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512CD[bit 28] = 1.

Similarly, the detection procedure for enumerating 512-bit instructions reported by AVX512DW follows the same flow.

## 2.3 DETECTION OF INTEL AVX-512 INSTRUCTION GROUPS OPERATING AT 256 AND 128-BIT VECTOR LENGTHS

For each of the 512-bit instruction groups in the Intel AVX-512 family listed in Table 2-1, EVEX encoding scheme may support a vast majority of these instructions operating at 256-bit or 128-bit (if applicable) vector lengths. This encoding support for vector lengths smaller than 512-bits is indicated by CPUID.(EAX=07H, ECX=0):EBX[bit 31], abbreviated as AVX512VL.

The AVX512VL flag alone is never sufficient to determine a given Intel AVX-512 instruction may be encoded at vector lengths smaller than 512 bits. Software must use the procedure described in Figure 2-3 and Table 2-2:



**Figure 2-3. Procedural Flow of Application Detection of Intel AVX-512 Instructions Operating at Vector Lengths < 512**

To illustrate the procedure described in Figure 2-3 and Table 2-2 for software to use EVEX.256 encoded VPCONFLICT, the following sequence is strongly recommended.

- 1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use)
- 2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
- 3) Verify CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512CD[bit 28] = 1, and CPUID.0x7.0:EBX.AVX512VL[bit 31] = 1.

**Table 2-2. Feature flag Collection Required of 256/128 Bit Vector Lengths for Each Instruction Group**

| Usage of 256/128 Vector Lengths | Feature Flag Collection to Verify |
|---------------------------------|-----------------------------------|
| AVX512F                         | AVX512F & AVX512VL                |
| AVX512CD                        | AVX512F & AVX512CD & AVX512VL     |
| AVX512DQ                        | AVX512F & AVX512DQ & AVX512VL     |
| AVX512BW                        | AVX512F & AVX512BW & AVX512VL     |
| AVX512FMA                       | AVX512F & AVX512FMA & AVX512VL    |
| AVX512VBMI                      | AVX512F & AVX512VBMI & AVX512VL   |

In some specific cases, AVX512VL may only support EVEX.256 encoding but not EVEX.128. These are listed in Table 2-3.

**Table 2-3. Instruction Mnemonics That Do Not Support EVEX.128 Encoding**

| Instruction Group | Instruction Mnemonics Supporting EVEX.256 Only Using AVX512VL  |
|-------------------|--|
| AVX512F           | VBROADCASTSD, VBROADCASTF32X4, VEXTRACTI32X4, VINSERTF32X4, VINSERTI32X4, VPERMD, VPERMPD, VPERMPS, VPERMQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2 |
| AVX512CD          |  |
| AVX512DQ          | VBROADCASTF32X2, VBROADCASTF64X2, VBROADCASTI32X4, VBROADCASTI64X2, VEXTRACTI64X2, VINSERTF64X2, VINSERTI64X2,   |
| AVX512BW          |  |

## 2.4 ACCESSING XMM, YMM AND ZMM REGISTERS

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAX\_VL-1:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register.

Upper bits of YMM registers (255:128) can be read and written by many instructions with a VEX.256 prefix.

XSAVE and XRSTOR may be used to save and restore the upper bits of the YMM registers.

The lower 256 bits of a ZMM register are aliased to the corresponding YMM register. Legacy SSE instructions (i.e. SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAX\_VL-1:128) of the ZMM registers, where MAX\_VL is maximum vector length (currently 512 bits). AVX and FMA instructions with a VEX prefix and vector length of 128-bits zero the upper 384 bits of the ZMM register, while VEX prefix and vector length of 256-bits zeros the upper 256 bits of the ZMM register.

Upper bits of ZMM registers (511:256) can be read and written by instructions with an EVEX.512 prefix.

## 2.5 ENHANCED VECTOR PROGRAMMING ENVIRONMENT USING EVEX ENCODING

EVEX-encoded AVX-512 instructions support an enhanced vector programming environment. The enhanced vector programming environment uses the combination of EVEX bit-field encodings and a set of eight opmask registers to provide the following capabilities:

- Conditional vector processing of EVEX-encoded instruction. Opmask registers k1 through k7 can be used to conditionally govern the per-data-element computational operation and the per-element updates to the



destination operand of an AVX-512 Foundation instruction. Each bit of the opmask register governs one vector element operation (a vector element can be of 32 bits or 64 bits).

- In addition to providing predication control on vector instructions via EVEX bit-field encoding, the opmask registers can also be used similarly to general-purpose registers as source/destination operands using modR/M encoding for non-mask-related instructions. In this case, an opmask register k0 through k7 can be selected.
- In 64-bit mode, 32 vector registers can be encoded using EVEX prefix.
- Broadcast may be supported for some instructions on the operand that can be encoded as a memory vector. The data elements of a memory vector may be conditionally fetched or written to, and the vector size is dependent on the data transformation function.
- Flexible rounding control for register-to-register flavor of EVEX encoded 512-bit and scalar instructions. Four rounding modes are supported by direct encoding within the EVEX prefix overriding MXCSR settings.
- Broadcast of one element to the rest of the destination vector register.
- Compressed 8-bit displacement encoding scheme to increase the instruction encoding density for instructions that normally require disp32 syntax.

### 2.5.1 OPMASK Register to Predicate Vector Data Processing

AVX-512 instructions using EVEX encodes a predicate operand to conditionally control per-element computational operation and updating of result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers of size MAX\_KL (64-bit). Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operand. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand. Note also that a predicate operand can be used to enable memory fault-suppression for some instructions with a memory operand (source or destination).

As a predicate operand, the opmask registers contain one bit to govern the operation/update to each data element of a vector register. In general, opmask registers can support instructions with element sizes: single-precision floating-point (float32), integer doubleword (int32), double-precision floating-point (float64), integer quadword (int64). The length of a opmask register, MAX\_KL, is sufficient to handle up to 64 elements with one bit per element, i.e. 64 bits. Masking is supported in most of the AVX-512 instructions. For a given vector length, each instruction accesses only the number of least significant mask bits that are needed based on its data type. For example, AVX-512 Foundation instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 least significant bits of the opmask register.

An opmask register affects an AVX-512 instruction at per-element granularity. So, any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in AVX-512 obeys the following properties:

- The instruction's operation is not performed for an element if the corresponding opmask bit is not set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value must be preserved (merging-masking) or it must be zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a versatile construct to implement control-flow predication as the mask in effect provides a merging behavior for AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior is provided to remove the implicit dependency on the old value when it is not needed.

Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory only accept merging-masking.

It's important to note that the per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- generated as a result of a vector instruction (e.g. CMP)
- loaded from memory
- loaded from GPR register
- or modified by mask-to-mask operations

Opmask registers can be used for purposes outside of predication. For example, they can be used to manipulate sparse sets of elements from a vector or used to set the EFLAGS based on the 0/0xFFFFFFFF/other status of the OR of two opmask registers.

### 2.5.1.1 Opmask Register K0

The only exception to the opmask rules described above is that opmask k0 can not be used as a predicate operand. Opmask k0 cannot be encoded as a predicate operand for a vector operation; the encoding value that would select opmask k0 will instead selects an implicit opmask value of 0xFFFFFFFF, thereby effectively disabling masking. Opmask register k0 can still be used for any instruction that takes opmask register(s) as operand(s) (either source or destination).

Note that certain instructions implicitly use the opmask as an extra destination operand. In such cases, trying to use the “no mask” feature will translate into a #UD fault being raised.

### 2.5.1.2 Example of Opmask Usages

The example below illustrates predicated vector add operation and predicated updates of added results into the destination operand. The initial state of vector registers zmm0, zmm1, and zmm2 and k3 are:

```

MSB.....LSB

zmm0 =
[ 0x00000003 0x00000002 0x00000001 0x00000000 ] (bytes 15 through 0)
[ 0x00000007 0x00000006 0x00000005 0x00000004 ] (bytes 31 through 16)
[ 0x0000000B 0x0000000A 0x00000009 0x00000008 ] (bytes 47 through 32)
[ 0x0000000F 0x0000000E 0x0000000D 0x0000000C ] (bytes 63 through 48)

zmm1 =
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 15 through 0)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 31 through 16)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 47 through 32)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 63 through 48)

zmm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA ] (bytes 15 through 0)
[ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ] (bytes 31 through 16)
[ 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC ] (bytes 47 through 32)
[ 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)

k3 = 0x8F03 (1000 1111 0000 0011)
    
```

An opmask register serving as a predicate operand is expressed as a curly-braces-enclosed decorator following the first operand in the Intel assembly syntax. Given this state, we will execute the following instruction:

```
vpaddq zmm2 {k3}, zmm0, zmm1
```

The `vpadd` instruction performs 32-bit integer additions on each data element conditionally based on the corresponding bit value in the predicate operand `k3`. Since per-element operations are not operated if the corresponding bit of the predicate mask is not set, the intermediate result is:

```
[ ***** ***** 0x00000010 0x0000000F ] (bytes 15 through 0)
[ ***** ***** ***** ***** ] (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E ***** ***** ***** ] (bytes 63 through 48)
```

where "\*\*\*\*\*" indicates that no operation is performed.

This intermediate result is then written into the destination vector register, `zmm2`, using the opmask register `k3` as the writemask, producing the following final result:

```
zmm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0x00000010 0x0000000F ] (bytes 15 through 0)
[ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ] (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)
```

Note that for a 64-bit instruction (say `vaddpd`), only the 8 LSB of mask `k3` (0x03) would be used to identify the predicate operation on each one of the 8 elements of the source/destination vectors.

## 2.5.2 OpMask Instructions

AVX-512 Foundation instructions provide a collection of opmask instructions that allow programmers to set, copy, or operate on the contents of a given opmask register. There are three types of opmask instructions:

- **Mask read/write instructions:** These instructions move data between a general-purpose integer register or memory and an opmask mask register, or between two opmask registers. For example:
  - `kmovw k1, ebx`; move lower 16 bits of `ebx` to `k1`.
- **Flag instructions:** This category, consisting of instructions that modify EFLAGS based on the content of opmask registers.
  - `kortestw k1, k2`; OR registers `k1` and `k2` and updated EFLAGS accordingly.
- **Mask logical instructions:** These instructions perform standard bitwise logical operations between opmask registers.
  - `kandw k1, k2, k3`; AND lowest 16 bits of registers `k2` and `k3`, leaving the result in `k1`.

## 2.5.3 Broadcast

EVEX encoding provides a bit-field to encode data broadcast for some load-op instructions, i.e. instructions that load data from memory and perform some computational or data movement operation. A source element from memory can be broadcast (repeated) across all the elements of the effective source operand (up to 16 times for 32-bit data element, up to 8 times for 64-bit data element). This is useful when we want to reuse the same scalar operand for all the operations in a vector instruction. Broadcast is only enabled on instructions with an element size of 32 bits or 64 bits. Byte and word instructions do not support embedded broadcast.

The functionality of data broadcast is expressed as a curly-braces-enclosed decorator following the last register/memory operand in the Intel assembly syntax.

For instance:

```
vmulps zmm1, zmm2, [rax] {1to16}
```

The {1to16} primitive loads one float32 (single precision) element from memory, replicates it 16 times to form a vector of 16 32-bit floating-point elements, multiplies the 16 float32 elements with the corresponding elements in the first source operand vector, and put each of the 16 results into the destination operand.

AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

```
vmovaps [rax] {k3}, zmm19
```

In contrast, the k3 opmask register is used as the predicate operand in the above example. Only the store operation on data elements corresponding to the non-zero bits in k3 will be performed.

### 2.5.4 STATIC ROUNDING MODE AND SUPPRESS ALL EXCEPTIONS

In previous SIMD instruction extensions, rounding control is generally specified in MXCSR, with a handful of instructions providing per-instruction rounding override via encoding fields within the imm8 operand. AVX-512 offers a more flexible encoding attribute to override MXCSR-based rounding control for floating-pointing instruction with rounding semantic. This rounding attribute embedded in the EVEX prefix is called Static (per instruction) Rounding Mode or Rounding Mode override. This attribute allows programmers to statically apply a specific arithmetic rounding mode irrespective of the value of RM bits in MXCSR. It is available only to register-to-register flavors of EVEX-encoded floating-point instructions with rounding semantic. The differences between these three rounding control interfaces are summarized in Table 2-4.

**Table 2-4. Characteristics of Three Rounding Control Interfaces**

| Rounding Interface            | Static Rounding Override         | Imm8 Embedded Rounding Override              | MXCSR Rounding Control            |
|-------------------------------|----------------------------------|--|-----------------------------------|
| Semantic Requirement          | FP rounding                      | FP rounding                                  | FP rounding                       |
| Prefix Requirement            | EVEX.B = 1                       | NA   | NA                                |
| Rounding Control              | EVEX.L'L                         | IMM8[1:0] or MXCSR.RC (depending on IMM8[2]) | MXCSR.RC                          |
| Suppress All Exceptions (SAE) | Implied                          | no   | no                                |
| SIMD FP Exception #XF         | All suppressed                   | Can raise #I, #P (unless SPE is set)         | MXCSR masking controls            |
| MXCSR flag update             | No                               | yes (except PE if SPE is set)                | Yes                               |
| Precedence                    | Above MXCSR.RC                   | Above EVEX.L'L                               | Default                           |
| Scope                         | 512-bit, reg-reg, Scalar reg-reg | ROUNDPx, ROUNDSx, VCVTPS2PH, VRNDSCALExx     | All SIMD operands, vector lengths |

The static rounding-mode override in AVX-512 also implies the “suppress-all-exceptions” (SAE) attribute. The SAE effect is as if all the MXCSR mask bits are set, and none of the MXCSR flags will be updated. Using static rounding-mode via EVEX without SAE is not supported.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In such a case, vector length is assumed to be MAX\_VL (512-bit in case of AVX-512 packed vector instructions) or 128-bit for scalar instructions. Table 2-5 summarizes the possible static rounding-mode assignments in AVX-512 instructions.

Note that some instructions already allow to specify the rounding mode statically via immediate bits. In such case, the immediate bits take precedence over the embedded rounding mode (in the same vein that they take precedence over whatever MXCSR.RM says).

Table 2-5. Static Rounding Mode

| Function | Description                        |
|----------|------------------------------------|
| {rn-sae} | Round to nearest (even) + SAE      |
| {rd-sae} | Round down (toward -inf) + SAE     |
| {ru-sae} | Round up (toward +inf) + SAE       |
| {rz-sae} | Round toward zero (Truncate) + SAE |

An example of use would be in the following instructions:

```
vaddps zmm7 {k6}, zmm2, zmm4, {rd-sae}
```

Which would perform the single-precision floating-point addition of vectors zmm2 and zmm4 with round-towards-minus-infinity, leaving the result in vector zmm7 using k6 as conditional writemask.

Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

Examples of instructions instances where the static rounding-mode is not allowed would be:

```
; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2, 0x00
```

```
; instructions with memory operands
vmulps zmm7 {k6}, zmm2, [rax], {rd-sae}
```

## 2.5.5 Compressed Disp8\*N Encoding

EVEX encoding supports a new displacement representation that allows for a more compact encoding of memory addressing commonly used in unrolled code, where an 8-bit displacement can address a range exceeding the dynamic range of an 8-bit value. This compressed displacement encoding is referred to as disp8\*N, where N is a constant implied by the memory operation characteristic of each instruction.

The compressed displacement is based on the assumption that the effective displacement (of a memory operand occurring in a loop) is a multiple of the granularity of the memory access of each iteration. Since the Base register in memory addressing already provides byte-granular resolution, the lower bits of the traditional disp8 operand becomes redundant, and can be implied from the memory operation characteristic.

The memory operation characteristics depend on the following:

- The destination operand is updated as a full vector, a single element, or multi-element tuples.
- The memory source operand (or vector source operand if the destination operand is memory) is fetched (or treated) as a full vector, a single element, or multi-element tuples.

For example,

```
vaddps zmm7, zmm2, disp8[membase + index*8]
```

The destination zmm7 is updated as a full 512-bit vector, and 64-bytes of data are fetched from memory as a full vector; the next unrolled iteration may fetch from memory in 64-byte granularity per iteration. There are 6 bits of lowest address that can be compressed, hence  $N = 2^6 = 64$ . The contribution of "disp8" to effective address calculation is  $64 * \text{disp8}$ .

```
vbroadcastf32x4 zmm7, disp8[membase + index*8]
```

In VBROADCASTF32x4, memory is fetched as a 4tuple of 4 32-bit entities. Hence the common lowest address bits that can be compressed is 4, corresponding to the 4tuple width of  $2^4 = 16$  bytes (4x32 bits). Therefore,  $N = 2^4$ .

For EVEX encoded instructions that update only one element in the destination, or source element is fetched individually, the number of lowest address bits that can be compressed is generally the width in bytes of the data element, hence  $N = 2^{(\text{width})}$ .

## 2.6 MEMORY ALIGNMENT

Memory alignment requirements on EVEX-encoded SIMD instructions are similar to VEX-encoded SIMD instructions. Memory alignment applies to EVEX-encoded SIMD instructions in three categories:

- Explicitly-aligned SIMD load and store instructions accessing 64 bytes of memory with EVEX prefix encoded vector length of 512 bits (e.g. VMOVAPD, VMOVAPS, VMOVDQA, etc.). These instructions always require memory address to be aligned on 64-byte boundary.
- Explicitly-unaligned SIMD load and store instructions accessing 64 bytes or less of data from memory (e.g. VMOVUPD, VMOVUPS, VMOVDQU, VMOVQ, VMOVD, etc.). These instructions do not require memory address to be aligned on natural vector-length byte boundary.
- Most arithmetic and data processing instructions encoded using EVEX support memory access semantics. When these instructions access from memory, there are no alignment restrictions.

Software may see performance penalties when unaligned accesses cross cacheline boundaries or vector-length naturally-aligned boundaries, so reasonable attempts to align commonly used data sets should continue to be pursued.

Atomic memory operation in Intel 64 and IA-32 architecture is guaranteed only for a subset of memory operand sizes and alignment scenarios. The guaranteed atomic operations are described in Section 7.1.1 of IA-32 Intel® Architecture Software Developer's Manual, Volumes 3A. AVX and FMA instructions do not introduce any new guaranteed atomic memory operations.

AVX-512 instructions may generate an #AC(0) fault on misaligned 4 or 8-byte memory references in Ring-3 when CR0.AM=1. 16, 32 and 64-byte memory references will not generate #AC(0) fault. See Table 2-7 for details.

Certain AVX-512 Foundation instructions always require 64-byte alignment (see the complete list of VEX and EVEX encoded instructions in Table 2-6). These instructions will #GP(0) if not aligned to 64-byte boundaries.

**Table 2-6. SIMD Instructions Requiring Explicitly Aligned Memory**

| Require 16-byte alignment | Require 32-byte alignment | Require 64-byte alignment* |
|---------------------------|---------------------------|----------------------------|
| (V)MOVDQA xmm, m128       | VMOVDQA ymm, m256         | VMOVDQA zmm, m512          |
| (V)MOVDQA m128, xmm       | VMOVDQA m256, ymm         | VMOVDQA m512, zmm          |
| (V)MOVAPS xmm, m128       | VMOVAPS ymm, m256         | VMOVAPS zmm, m512          |
| (V)MOVAPS m128, xmm       | VMOVAPS m256, ymm         | VMOVAPS m512, zmm          |
| (V)MOVAPD xmm, m128       | VMOVAPD ymm, m256         | VMOVAPD zmm, m512          |
| (V)MOVAPD m128, xmm       | VMOVAPD m256, ymm         | VMOVAPD m512, zmm          |
| (V)MOVNTDQA xmm, m128     | VMOVNTPS m256, ymm        | VMOVNTPS m512, zmm         |
| (V)MOVNTPS m128, xmm      | VMOVNTPD m256, ymm        | VMOVNTPD m512, zmm         |
| (V)MOVNTPD m128, xmm      | VMOVNTDQ m256, ymm        | VMOVNTDQ m512, zmm         |
| (V)MOVNTDQ m128, xmm      | VMOVNTDQA ymm, m256       | VMOVNTDQA zmm, m512        |

**Table 2-7. Instructions Not Requiring Explicit Memory Alignment**

|                      |                   |                   |
|----------------------|-------------------|-------------------|
| (V)MOVDQU xmm, m128  | VMOVDQU ymm, m256 | VMOVDQU zmm, m512 |
| (V)MOVDQU m128, m128 | VMOVDQU m256, ymm | VMOVDQU m512, zmm |
| (V)MOVUPS xmm, m128  | VMOVUPS ymm, m256 | VMOVUPS zmm, m512 |
| (V)MOVUPS m128, xmm  | VMOVUPS m256, ymm | VMOVUPS m512, zmm |
| (V)MOVUPD xmm, m128  | VMOVUPD ymm, m256 | VMOVUPD zmm, m512 |
| (V)MOVUPD m128, xmm  | VMOVUPD m256, ymm | VMOVUPD m512, zmm |

## 2.7 SIMD FLOATING-POINT EXCEPTIONS

AVX-512 instructions can generate SIMD floating-point exceptions (#XM) if embedded “suppress all exceptions” (SAE) in EVEX is not set. When SAE is not set, these instructions will respond to exception masks of MXCSR in the same way as VEX-encoded AVX instructions. When CR4.OSXMMEXCPT=0 any unmasked FP exceptions generate an Undefined Opcode exception (#UD).

## 2.8 INSTRUCTION EXCEPTION SPECIFICATION

Exception behavior of VEX-encoded AVX/AVX2 instructions are described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*. Exception behavior of AVX-512 Foundation instructions and additional 512-bit extensions are described in Section 4.10, “Exception Classifications of EVEX-Encoded instructions” and Section 4.11, “Exception Classifications of Opmask instructions”.

## 2.9 CPUID INSTRUCTION

### CPUID—CPU Identification

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

#### Description

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

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

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

Table 2-8 shows information returned, depending on the initial value loaded into the EAX register. Table 2-9 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value is entered for CPUID.EAX is invalid for a particular processor, the data for the highest basic information leaf is returned. For example, using the Intel Core 2 Duo E6850 processor, the following is true:

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

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

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

#### See also:

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

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

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



**Table 2-8. Information Returned by CPUID Instruction**

| Initial EAX Value   | Information Provided about the Processor |  |
|---|--|--|
| <i>Basic CPUID Information</i>  |  |  |
| 0H  | EAX<br>EBX<br>ECX<br>EDX                 | Maximum Input Value for Basic CPUID Information (see Table 2-9)<br>"Genu"<br>"ntel"<br>"inel"  |
| 01H   | EAX<br><br>EBX<br><br>ECX<br>EDX         | Version Information: Type, Family, Model, and Stepping ID (see Figure 2-4)<br><br>Bits 7-0: Brand Index<br>Bits 15-8: CLFLUSH line size (Value * 8 = cache line size in bytes)<br>Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*.<br>Bits 31-24: Initial APIC ID<br><br>Feature Information (see Figure 2-5 and Table 2-11)<br>Feature Information (see Figure 2-6 and Table 2-12)<br><b>NOTES:</b><br>* The nearest power-of-2 integer that is not smaller than EBX[23:16] is the maximum number of unique initial APIC IDs reserved for addressing different logical processors in a physical package.                     |
| 02H   | EAX<br>EBX<br>ECX<br>EDX                 | Cache and TLB Information (see Table 2-13)<br>Cache and TLB Information<br>Cache and TLB Information<br>Cache and TLB Information  |
| 03H   | EAX<br>EBX<br>ECX<br>EDX                 | Reserved.<br>Reserved.<br>Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)<br>Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)<br><b>NOTES:</b><br>Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.<br>See AP-485, <i>Intel Processor Identification and the CPUID Instruction</i> (Order Number 241618) for more information on PSN. |
| CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default). |  |  |
| <i>Deterministic Cache Parameters Leaf</i>  |  |  |
| 04H   | EAX                                      | <b>NOTES:</b><br>Leaf 04H output depends on the initial value in ECX.<br>See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 2-30."<br><br>Bits 4-0: Cache Type Field<br>0 = Null - No more caches<br>1 = Data Cache<br>2 = Instruction Cache<br>3 = Unified Cache<br>4-31 = Reserved  |

**Table 2-8. Information Returned by CPUID Instruction(Continued)**

| Initial EAX Value         | Information Provided about the Processor |  |
|---------------------------|--|--|
|                           |  | <p>Bits 7-5: Cache Level (starts at 1)<br/>                     Bits 8: Self Initializing cache level (does not need SW initialization)<br/>                     Bits 9: Fully Associative cache</p> <p>Bits 13-10: Reserved<br/>                     Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache*, **<br/>                     Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package*, ***, ****</p> <p>EBX Bits 11-00: L = System Coherency Line Size*<br/>                     Bits 21-12: P = Physical Line partitions*<br/>                     Bits 31-22: W = Ways of associativity*</p> <p>ECX Bits 31-00: S = Number of Sets*</p> <p>EDX Bit 0: WBINVD/INVD behavior on lower level caches<br/>                     Bit 10: Write-Back Invalidate/Invalidate<br/>                     0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache<br/>                     1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.<br/>                     Bit 1: Cache Inclusiveness<br/>                     0 = Cache is not inclusive of lower cache levels.<br/>                     1 = Cache is inclusive of lower cache levels.<br/>                     Bit 2: Complex cache indexing<br/>                     0 = Direct mapped cache<br/>                     1 = A complex function is used to index the cache, potentially using all address bits.<br/>                     Bits 31-03: Reserved = 0</p> <p><b>NOTES:</b><br/>                     * Add one to the return value to get the result.<br/>                     ** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache<br/>                     *** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.<br/>                     ****The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p> |
| <i>MONITOR/MWAIT Leaf</i> |  |  |
| 05H                       | EAX                                      | Bits 15-00: Smallest monitor-line size in bytes (default is processor’s monitor granularity)<br>Bits 31-16: Reserved = 0   |
|                           | EBX                                      | Bits 15-00: Largest monitor-line size in bytes (default is processor’s monitor granularity)<br>Bits 31-16: Reserved = 0  |
|                           | ECX                                      | Bits 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported<br>Bits 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled<br>Bits 31 - 02: Reserved   |

Table 2-8. Information Returned by CPUID Instruction(Continued)

| Initial EAX Value                        | Information Provided about the Processor  |  |
|--|---|--|
|  | EDX   | Bits 03 - 00: Number of C0* sub C-states supported using MWait<br>Bits 07 - 04: Number of C1* sub C-states supported using MWait<br>Bits 11 - 08: Number of C2* sub C-states supported using MWait<br>Bits 15 - 12: Number of C3* sub C-states supported using MWait<br>Bits 19 - 16: Number of C4* sub C-states supported using MWait<br>Bits 31 - 20: Reserved = 0<br><b>NOTE:</b><br>* The definition of C0 through C4 states for MWait extension are processor-specific C-states, not ACPI C-states.   |
| <i>Thermal and Power Management Leaf</i> |   |  |
| 06H                                      | EAX   | Bits 00: Digital temperature sensor is supported if set<br>Bits 01: Intel Turbo Boost Technology is available<br>Bits 31 - 02: Reserved  |
|  | EBX   | Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor<br>Bits 31 - 04: Reserved   |
|  | ECX   | Bits 00: Hardware Coordination Feedback Capability (Presence of MCNT and ACNT MSRs). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of expected processor performance at frequency specified in CPUID Brand String<br>Bits 02 - 01: Reserved = 0<br>Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H)<br>Bits 31 - 04: Reserved = 0 |
|  | EDX   | Reserved = 0   |
| <i>Structured Extended feature Leaf</i>  |   |  |
| 07H                                      | <b>NOTES:</b><br>Leaf 07H main leaf (ECX = 0).<br>IF leaf 07H is not supported, EAX=EBX=ECX=EDX=0 |  |
|  | EAX   | Bits 31-00: Reports the maximum number sub-leaves that are supported in leaf 07H.  |



Table 2-8. Information Returned by CPUID Instruction(Continued)

| Initial EAX Value                                | Information Provided about the Processor  |   |
|--|---|---|
| <i>Architectural Performance Monitoring Leaf</i> |   |   |
| 0AH  | EAX   | Bits 07 - 00: Version ID of architectural performance monitoring<br>Bits 15- 08: Number of general-purpose performance monitoring counter per logical processor<br>Bits 23 - 16: Bit width of general-purpose, performance monitoring counter<br>Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events  |
|  | EBX   | Bit 00: Core cycle event not available if 1<br>Bit 01: Instruction retired event not available if 1<br>Bit 02: Reference cycles event not available if 1<br>Bit 03: Last-level cache reference event not available if 1<br>Bit 04: Last-level cache misses event not available if 1<br>Bit 05: Branch instruction retired event not available if 1<br>Bit 06: Branch mispredict retired event not available if 1<br>Bits 31- 07: Reserved = 0 |
|  | ECX   | Reserved = 0<br>Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1)<br>Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1)  |
|  | EDX   | Reserved = 0  |
| <i>Extended Topology Enumeration Leaf</i>        |   |   |
| 0BH  | <p><b>NOTES:</b></p> <p>Most of Leaf 0BH output depends on the initial value in ECX.<br/>The EDX output of leaf 0BH is always valid and does not vary with input value in ECX<br/>Output value in ECX[7:0] always equals input value in ECX[7:0].<br/>For sub-leaves that returns an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0<br/>If an input value N in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX &gt; N also return 0 in ECX[15:8]</p>   |   |
|  | EAX   | Bits 04-00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level.<br>Bits 31-5: Reserved.   |
|  | EBX   | Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**.<br>Bits 31- 16: Reserved.   |
|  | ECX   | Bits 07 - 00: Level number. Same value in ECX input<br>Bits 15 - 08: Level type***.<br>Bits 31 - 16: Reserved.  |
| EDX  | Bits 31- 00: x2APIC ID the current logical processor.<br><br><p><b>NOTES:</b></p> <p>* Software should use this field (EAX[4:0]) to enumerate processor topology of the system.</p> <p>** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.</p> <p>*** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding:<br/>0: invalid<br/>1: SMT<br/>2: Core<br/>3-255: Reserved</p> |   |

**Table 2-8. Information Returned by CPUID Instruction(Continued)**

| Initial EAX Value  | Information Provided about the Processor   |   |
|--|--|---|
| <i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>   |  |   |
| 0DH  | <p><b>NOTES:</b><br/>Leaf 0DH main leaf (ECX = 0).</p>   |   |
|  | EAX  | <p>Bits 31-00: Reports the valid bit fields of the lower 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XCRO is reserved.<br/>                     Bit 00: legacy x87<br/>                     Bit 01: 128-bit SSE<br/>                     Bit 02: 256-bit AVX<br/>                     Bits 04 - 03: MPX state<br/>                     Bit 07 - 05: AVX-512 state<br/>                     Bit 08: Used for IA32_XSS<br/>                     Bit 09: PKRU state<br/>                     Bits 31-10: Reserved</p> |
|  | EBX  | <p>Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCRO. May be different than ECX if some features at the end of the XSAVE save area are not enabled.</p>   |
|  | ECX  | <p>Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCRO.</p>   |
|  | EDX  | <p>Bit 31-0: Reports the valid bit fields of the upper 32 bits of the XCRO register. If a bit is 0, the corresponding bit field in XCRO is reserved</p>   |
| <i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>  |  |   |
|  | EAX  | <p>Bit 00: XSAVEOPT is available;<br/>Bits 31-1: Reserved</p>   |
|  | EBX  | Reserved  |
|  | ECX  | Reserved  |
|  | EDX  | Reserved  |
| <i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n &gt; 1)</i>  |  |   |
| 0DH  | <p><b>NOTES:</b><br/>Leaf 0DH output depends on the initial value in ECX.<br/>If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.<br/>Each valid sub-leaf index maps to a valid bit in the XCRO register starting at bit position 2</p> |   |
|  | EAX  | <p>Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, <i>n</i>. This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*.</p>  |
|  | EBX  | <p>Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area.<br/>This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*.</p>  |
|  | ECX  | <p>Bit 0 is set if the sub-leaf index, <i>n</i>, maps to a valid bit in the IA32_XSS MSR and bit 0 is clear if <i>n</i> maps to a valid bit in XCRO. Bits 31-1 are reserved. This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*.</p>   |
|  | EDX  | <p>This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*; otherwise it is reserved.</p>   |
| <p>*The highest valid sub-leaf index, <i>n</i>, is<br/>(POPCNT(CPUID.(EAX=0D, ECX=0);EAX) + POPCNT(CPUID.(EAX=0D, ECX=0);EDX) - 1)</p> |  |   |
| <i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>  |  |   |
| 14H  | <p><b>NOTES:</b><br/>Leaf 14H main leaf (ECX = 0).</p>   |   |

Table 2-8. Information Returned by CPUID Instruction(Continued)

| Initial EAX Value                           | Information Provided about the Processor |  |
|---|--|--|
|   | EAX                                      | Bits 31-0: Reports the maximum number sub-leaves that are supported in leaf 14H.   |
|   | EBX                                      | Bit 00: If 1, Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed.<br>Bits 31- 01: Reserved   |
|   | ECX                                      | Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed.<br>Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.<br>Bit 30:02: Reserved<br>Bit 31: If 1, Generated packets which contain IP payloads have LIP values, which include the CS base component.   |
|   | EDX                                      | Bits 31- 00: Reserved  |
| <i>Processor Frequency Information Leaf</i> |  |  |
| 16H   | EAX                                      | Bits 15:0: Processor Base Frequency (in MHz).<br>Bits 31:16: Reserved =0   |
|   | EBX                                      | Bits 15:0: Maximum Frequency (in MHz).<br>Bits 31:16: Reserved = 0   |
|   | ECX                                      | Bits 15:0: Bus (Reference) Frequency (in MHz).<br>Bits 31:16: Reserved = 0   |
|   | EDX                                      | Reserved   |
|   |  | <b>NOTES:</b><br>* Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.<br><br>While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported. |
| <i>Extended Function CPUID Information</i>  |  |  |
| 80000000H                                   | EAX                                      | Maximum Input Value for Extended Function CPUID Information (see Table 2-9).   |
|   | EBX                                      | Reserved   |
|   | ECX                                      | Reserved   |
|   | EDX                                      | Reserved   |
| 80000001H                                   | EAX                                      | Extended Processor Signature and Feature Bits.   |
|   | EBX                                      | Reserved   |
|   | ECX                                      | Bit 0: LAHF/SAHF available in 64-bit mode<br>Bits 4-1: Reserved<br>Bit 5: LZCNT available<br>Bits 7-6 Reserved<br>Bit 8: PREFETCHW<br>Bits 31-9: Reserved  |

**Table 2-8. Information Returned by CPUID Instruction(Continued)**

| Initial EAX Value | Information Provided about the Processor |   |
|-------------------|--|---|
|                   | EDX                                      | Bits 10-0: Reserved<br>Bit 11: SYSCALL/SYSRET available (when in 64-bit mode)<br>Bits 19-12: Reserved = 0<br>Bit 20: Execute Disable Bit available<br>Bits 28-21: Reserved = 0<br>Bit 29: Intel® 64 Architecture available if 1<br>Bits 31-30: Reserved = 0   |
| 8000002H          | EAX<br>EBX<br>ECX<br>EDX                 | Processor Brand String<br>Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued  |
| 8000003H          | EAX<br>EBX<br>ECX<br>EDX                 | Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued  |
| 8000004H          | EAX<br>EBX<br>ECX<br>EDX                 | Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued  |
| 8000005H          | EAX<br>EBX<br>ECX<br>EDX                 | Reserved = 0<br>Reserved = 0<br>Reserved = 0<br>Reserved = 0  |
| 8000006H          | EAX<br>EBX<br><br>ECX<br><br>EDX         | Reserved = 0<br>Reserved = 0<br>Bits 7-0: Cache Line size in bytes<br>Bits 15-12: L2 Associativity field *<br>Bits 31-16: Cache size in 1K units<br>Reserved = 0<br><b>NOTES:</b><br>* L2 associativity field encodings:<br>00H - Disabled<br>01H - Direct mapped<br>02H - 2-way<br>04H - 4-way<br>06H - 8-way<br>08H - 16-way<br>0FH - Fully associative |
| 8000007H          | EAX<br>EBX<br>ECX<br>EDX                 | Reserved = 0<br>Reserved = 0<br>Reserved = 0<br>Reserved = 0  |
| 8000008H          | EAX                                      | Virtual/Physical Address size<br>Bits 7-0: #Physical Address Bits*<br>Bits 15-8: #Virtual Address Bits<br>Bits 31-16: Reserved = 0  |



**Table 2-8. Information Returned by CPUID Instruction(Continued)**

| Initial EAX Value | Information Provided about the Processor  |              |
|-------------------|---|--------------|
|                   | EBX   | Reserved = 0 |
|                   | ECX   | Reserved = 0 |
|                   | EDX   | Reserved = 0 |
|                   | <b>NOTES:</b><br>* If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field. |              |

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

When CPUID executes with EAX set to 0H, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 2-9) and is processor specific.

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

EBX ← 756e6547h (\* "Genu", with G in the low 4 bits of BL \*)

EDX ← 49656e69h (\* "inel", with i in the low 4 bits of DL \*)

ECX ← 6c65746eh (\* "ntel", with n in the low 4 bits of CL \*)

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

When CPUID executes with EAX set to 0H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 2-9) and is processor specific.

**Table 2-9. Highest CPUID Source Operand for Intel 64 and IA-32 Processors**

| Intel 64 or IA-32 Processors                                      | Highest Value in EAX  |                               |
|---|-----------------------|-------------------------------|
|   | Basic Information     | Extended Function Information |
| Earlier Intel486 Processors                                       | CPUID Not Implemented | CPUID Not Implemented         |
| Later Intel486 Processors and Pentium Processors                  | 01H                   | Not Implemented               |
| Pentium Pro and Pentium II Processors, Intel® Celeron® Processors | 02H                   | Not Implemented               |
| Pentium III Processors  | 03H                   | Not Implemented               |
| Pentium 4 Processors  | 02H                   | 80000004H                     |
| Intel Xeon Processors   | 02H                   | 80000004H                     |
| Pentium M Processor   | 02H                   | 80000004H                     |
| Pentium 4 Processor supporting Hyper-Threading Technology         | 05H                   | 80000008H                     |
| Pentium D Processor (8xx)   | 05H                   | 80000008H                     |
| Pentium D Processor (9xx)   | 06H                   | 80000008H                     |
| Intel Core Duo Processor  | 0AH                   | 80000008H                     |
| Intel Core 2 Duo Processor  | 0AH                   | 80000008H                     |
| Intel Xeon Processor 3000, 5100, 5300 Series                      | 0AH                   | 80000008H                     |

**Table 2-9. Highest CPUID Source Operand for Intel 64 and IA-32 Processors (Continued)**

| Intel 64 or IA-32 Processors                             | Highest Value in EAX |                               |
|--|----------------------|-------------------------------|
|  | Basic Information    | Extended Function Information |
| Intel Xeon Processor 3000, 5100, 5200, 5300, 5400 Series | 0AH                  | 80000008H                     |
| Intel Core 2 Duo Processor 8000 Series                   | 0DH                  | 80000008H                     |
| Intel Xeon Processor 5200, 5400 Series                   | 0AH                  | 80000008H                     |

**IA32\_BIOS\_SIGN\_ID Returns Microcode Update Signature**

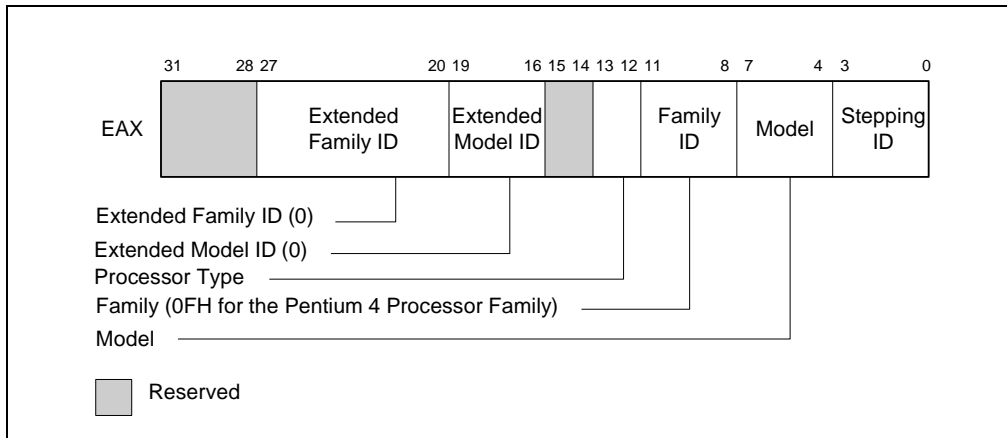
For processors that support the microcode update facility, the IA32\_BIOS\_SIGN\_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

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

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 2-4). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

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

See Table 2-10 for available processor type values. Stepping IDs are provided as needed.



**Figure 2-4. Version Information Returned by CPUID in EAX**

**Table 2-10. Processor Type Field**

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

**NOTE**

See "Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, and Chapter 16 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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

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

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

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

**INPUT EAX = 01H: Returns Additional Information in EBX**

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

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

**INPUT EAX = 01H: Returns Feature Information in ECX and EDX**

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

- Figure 2-5 and Table 2-11 show encodings for ECX.
- Figure 2-6 and Table 2-12 show encodings for EDX.

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

**NOTE**

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

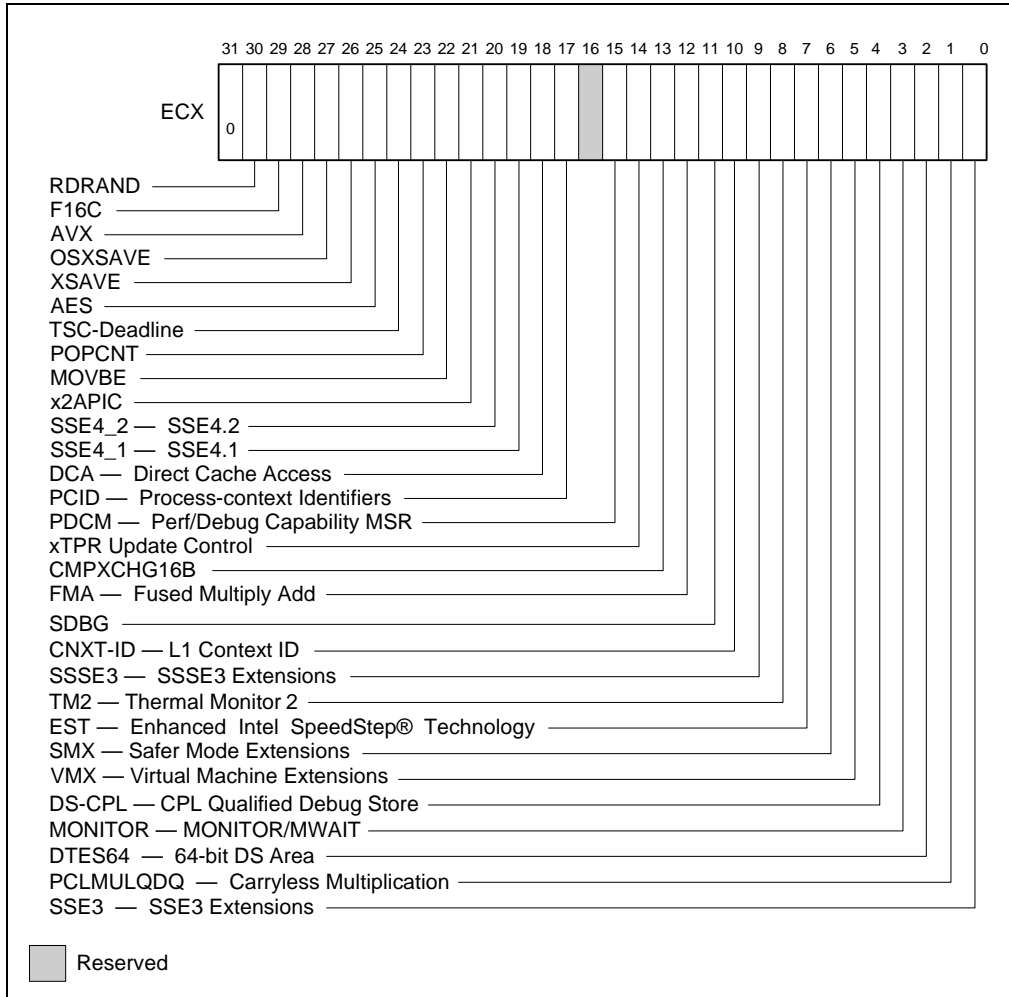


Figure 2-5. Feature Information Returned in the ECX Register

Table 2-11. Feature Information Returned in the ECX Register

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

Table 2-11. Feature Information Returned in the ECX Register (Continued)

| Bit # | Mnemonic            | Description  |
|-------|---------------------|--|
| 10    | CNXT-ID             | <b>L1 Context ID.</b> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details. |
| 11    | Reserved            | Reserved   |
| 12    | FMA                 | A value of 1 indicates the processor supports FMA extensions using YMM state.  |
| 13    | CMPXCHG16B          | <b>CMPXCHG16B Available.</b> A value of 1 indicates that the feature is available.   |
| 14    | xTPR Update Control | <b>xTPR Update Control.</b> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].  |
| 15    | PDCM                | <b>Perfmon and Debug Capability:</b> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.  |
| 16    | Reserved            | Reserved   |
| 17    | PCID                | <b>Process-context identifiers.</b> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.   |
| 18    | DCA                 | A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.  |
| 19    | SSE4.1              | A value of 1 indicates that the processor supports SSE4.1.   |
| 20    | SSE4.2              | A value of 1 indicates that the processor supports SSE4.2.   |
| 21    | x2APIC              | A value of 1 indicates that the processor supports x2APIC feature.   |
| 22    | MOVBE               | A value of 1 indicates that the processor supports MOVBE instruction.  |
| 23    | POPCNT              | A value of 1 indicates that the processor supports the POPCNT instruction.   |
| 24    | TSC-Deadline        | A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.   |
| 25    | AES                 | A value of 1 indicates that the processor supports the AES instruction.  |
| 26    | XSAVE               | A value of 1 indicates that the processor supports the XFEATURE_ENABLED_MASK register and XSAVE/XRSTOR/XSETBV/XGETBV instructions to manage processor extended states.   |
| 27    | OSXSAVE             | A value of 1 indicates that the OS has enabled support for using XGETBV/XSETBV instructions to query processor extended states.  |
| 28    | AVX                 | A value of 1 indicates that processor supports AVX instructions operating on 256-bit YMM state, and three-operand encoding of 256-bit and 128-bit SIMD instructions.   |
| 29    | F16C                | A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.  |
| 30    | RDRAND              | A value of 1 indicates that processor supports RDRAND instruction.   |
| 31    | Not Used            | Always return 0.   |

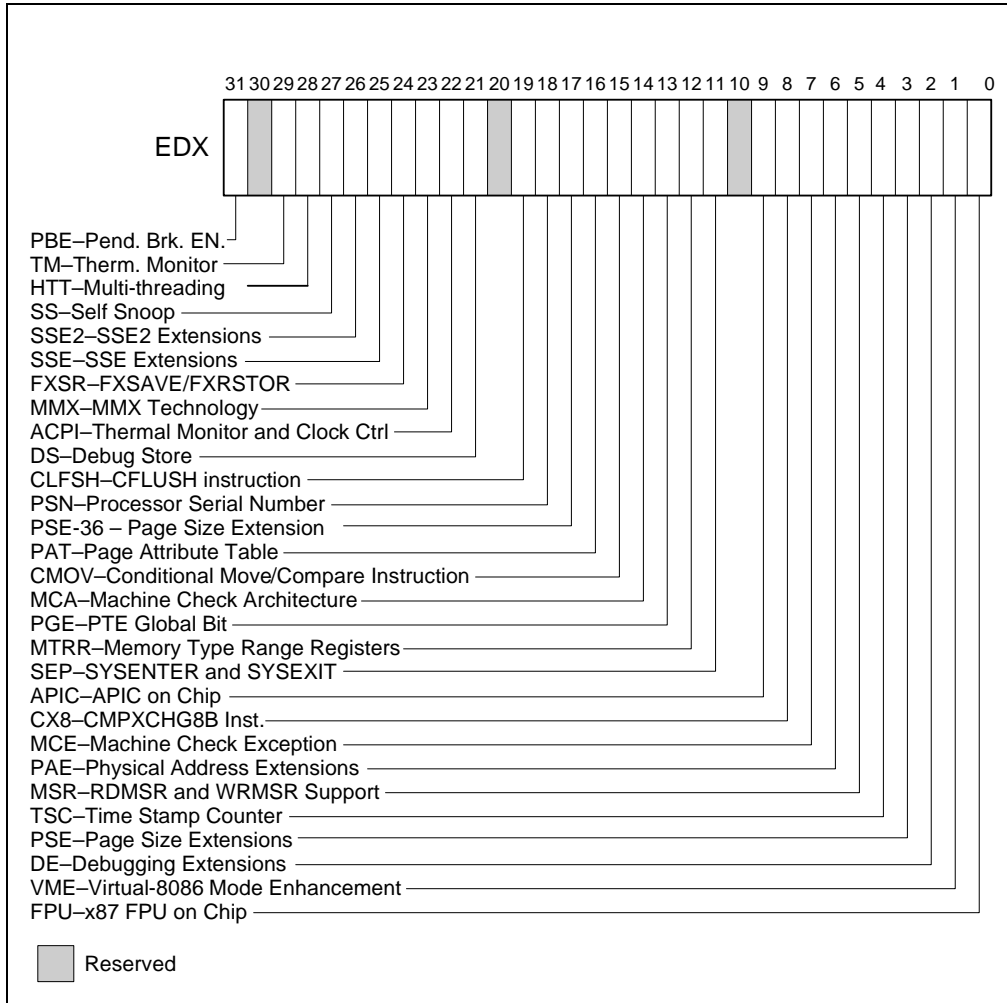


Figure 2-6. Feature Information Returned in the EDX Register

Table 2-12. More on Feature Information Returned in the EDX Register

| Bit # | Mnemonic | Description  |
|-------|----------|--|
| 0     | FPU      | <b>floating-point Unit On-Chip.</b> The processor contains an x87 FPU.   |
| 1     | VME      | <b>Virtual 8086 Mode Enhancements.</b> Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags. |
| 2     | DE       | <b>Debugging Extensions.</b> Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.  |
| 3     | PSE      | <b>Page Size Extension.</b> Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.  |
| 4     | TSC      | <b>Time Stamp Counter.</b> The RDTSC instruction is supported, including CR4.TSD for controlling privilege.  |
| 5     | MSR      | <b>Model Specific Registers RDMSR and WRMSR Instructions.</b> The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.   |

Table 2-12. More on Feature Information Returned in the EDX Register(Continued)

| Bit # | Mnemonic | Description  |
|-------|----------|--|
| 6     | PAE      | <b>Physical Address Extension.</b> Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific.   |
| 7     | MCE      | <b>Machine Check Exception.</b> Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature. |
| 8     | CX8      | <b>CMPXCHG8B Instruction.</b> The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).  |
| 9     | APIC     | <b>APIC On-Chip.</b> The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).  |
| 10    | Reserved | Reserved   |
| 11    | SEP      | <b>SYSENTER and SYSEXIT Instructions.</b> The SYSENTER and SYSEXIT and associated MSR are supported.   |
| 12    | MTRR     | <b>Memory Type Range Registers.</b> MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.  |
| 13    | PGE      | <b>PTE Global Bit.</b> The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.   |
| 14    | MCA      | <b>Machine Check Architecture.</b> The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.   |
| 15    | CMOV     | <b>Conditional Move Instructions.</b> The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported  |
| 16    | PAT      | <b>Page Attribute Table.</b> Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.  |
| 17    | PSE-36   | <b>36-Bit Page Size Extension.</b> Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.  |
| 18    | PSN      | <b>Processor Serial Number.</b> The processor supports the 96-bit processor identification number feature and the feature is enabled.  |
| 19    | CLFSH    | <b>CLFLUSH Instruction.</b> CLFLUSH Instruction is supported.  |
| 20    | Reserved | Reserved   |
| 21    | DS       | <b>Debug Store.</b> The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 17, "Debug, Branch Profile, TSC, and Quality of Service," in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A</i> ).  |
| 22    | ACPI     | <b>Thermal Monitor and Software Controlled Clock Facilities.</b> The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.   |
| 23    | MMX      | <b>Intel MMX Technology.</b> The processor supports the Intel MMX technology.  |
| 24    | FXSR     | <b>FXSAVE and FXRSTOR Instructions.</b> The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating-point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.  |

**Table 2-12. More on Feature Information Returned in the EDX Register(Continued)**

| Bit # | Mnemonic | Description   |
|-------|----------|---|
| 25    | SSE      | <b>SSE.</b> The processor supports the SSE extensions.  |
| 26    | SSE2     | <b>SSE2.</b> The processor supports the SSE2 extensions.  |
| 27    | SS       | <b>Self Snoop.</b> The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.   |
| 28    | HTT      | <b>Multi-Threading.</b> The physical processor package is capable of supporting more than one logical processor.  |
| 29    | TM       | <b>Thermal Monitor.</b> The processor implements the thermal monitor automatic thermal control circuitry (TCC).   |
| 30    | Reserved | Reserved  |
| 31    | PBE      | <b>Pending Break Enable.</b> The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability. |

**INPUT EAX = 02H: Cache and TLB Information Returned in EAX, EBX, ECX, EDX**

When CPUID executes with EAX set to 02H, the processor returns information about the processor’s internal caches and TLBs in the EAX, EBX, ECX, and EDX registers.

The encoding is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 02H to get a complete description of the processor’s caches and TLBs. The first member of the family of Pentium 4 processors will return a 01H.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 2-13 shows the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache or TLB types. The descriptors may appear in any order.

**Table 2-13. Encoding of Cache and TLB Descriptors**

| Descriptor Value | Cache or TLB Description  |
|------------------|---|
| 00H              | Null descriptor   |
| 01H              | Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries                         |
| 02H              | Instruction TLB: 4 MByte pages, 4-way set associative, 2 entries                          |
| 03H              | Data TLB: 4 KByte pages, 4-way set associative, 64 entries                                |
| 04H              | Data TLB: 4 MByte pages, 4-way set associative, 8 entries                                 |
| 05H              | Data TLB1: 4 MByte pages, 4-way set associative, 32 entries                               |
| 06H              | 1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size           |
| 08H              | 1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size          |
| 0AH              | 1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size                  |
| 0BH              | Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries                          |
| 0CH              | 1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size                 |
| 22H              | 3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector |
| 23H              | 3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 25H              | 3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 29H              | 3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 2CH              | 1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size                 |



**Table 2-13. Encoding of Cache and TLB Descriptors (Continued)**

| Descriptor Value | Cache or TLB Description   |
|------------------|--|
| 30H              | 1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size   |
| 40H              | No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache   |
| 41H              | 2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size  |
| 42H              | 2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size  |
| 43H              | 2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size  |
| 44H              | 2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size   |
| 45H              | 2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size   |
| 46H              | 3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size   |
| 47H              | 3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size   |
| 49H              | 3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H);<br>2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size |
| 4AH              | 3rd-level cache: 6MByte, 12-way set associative, 64 byte line size   |
| 4BH              | 3rd-level cache: 8MByte, 16-way set associative, 64 byte line size   |
| 4CH              | 3rd-level cache: 12MByte, 12-way set associative, 64 byte line size  |
| 4DH              | 3rd-level cache: 16MByte, 16-way set associative, 64 byte line size  |
| 4EH              | 2nd-level cache: 6MByte, 24-way set associative, 64 byte line size   |
| 50H              | Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries  |
| 51H              | Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries   |
| 52H              | Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries   |
| 56H              | Data TLB0: 4 MByte pages, 4-way set associative, 16 entries  |
| 57H              | Data TLB0: 4 KByte pages, 4-way associative, 16 entries  |
| 5BH              | Data TLB: 4 KByte and 4 MByte pages, 64 entries  |
| 5CH              | Data TLB: 4 KByte and 4 MByte pages, 128 entries   |
| 5DH              | Data TLB: 4 KByte and 4 MByte pages, 256 entries   |
| 60H              | 1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size   |
| 66H              | 1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size  |
| 67H              | 1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size   |
| 68H              | 1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size   |
| 70H              | Trace cache: 12 K- $\mu$ op, 8-way set associative   |
| 71H              | Trace cache: 16 K- $\mu$ op, 8-way set associative   |
| 72H              | Trace cache: 32 K- $\mu$ op, 8-way set associative   |
| 78H              | 2nd-level cache: 1 MByte, 4-way set associative, 64byte line size  |
| 79H              | 2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 7AH              | 2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 7BH              | 2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 7CH              | 2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector   |
| 7DH              | 2nd-level cache: 2 MByte, 8-way set associative, 64byte line size  |
| 7FH              | 2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size   |
| 82H              | 2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size   |
| 83H              | 2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size   |
| 84H              | 2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size   |

**Table 2-13. Encoding of Cache and TLB Descriptors (Continued)**

| Descriptor Value | Cache or TLB Description  |
|------------------|---|
| 85H              | 2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size        |
| 86H              | 2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size      |
| 87H              | 2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size        |
| B0H              | Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries        |
| B1H              | Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries |
| B3H              | Data TLB: 4 KByte pages, 4-way set associative, 128 entries               |
| B4H              | Data TLB1: 4 KByte pages, 4-way associative, 256 entries                  |
| F0H              | 64-Byte prefetching   |
| F1H              | 128-Byte prefetching  |

**Example 2-1. Example of Cache and TLB Interpretation**

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

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

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K- $\mu$ op, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

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

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

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 2-8.

The CPUID leaf 4 also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0H and use it as part of the topology enumeration algorithm described in Chapter

8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

#### INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 2-8.

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

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 2-8.

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

When CPUID executes with EAX set to 07H and ECX = 0H, the processor returns information about the maximum number of sub-leaves that contain extended feature flags. See Table 2-8.

When CPUID executes with EAX set to 07H and ECX = n (n > 1 and less than the number of non-zero bits in CPUID.(EAX=07H, ECX= 0H).EAX, the processor returns information about extended feature flags. See Table 2-8. In sub-leaf 0, only EAX has the number of sub-leaves. In sub-leaf 0, EBX, ECX & EDX all contain extended feature flags.

**Table 2-14. Structured Extended Feature Leaf, Function 0, EBX Register**

| Bit # | Mnemonic   | Description   |
|-------|------------|---|
| 0     | RWFSGSBASE | A value of 1 indicates the processor supports RD/WR FSGSBASE instructions |
| 1-31  | Reserved   | Reserved  |

#### INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 2-8.

#### INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 2-8) is greater than Pn 0. See Table 2-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 17, “Debug, Branch Profile, TSC, and Quality of Service,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

#### INPUT EAX = 0BH: Returns Extended Topology Information

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

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

When CPUID executes with EAX set to 0DH and ECX = 0H, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 2-8.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1 and less than the number of non-zero bits in CPUID.(EAX=0DH, ECX= 0H).EDX), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 2-8.

#### **INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information**

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 2-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 1 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EDX), the processor returns information about packet generation in Intel Processor Trace. See Table 2-8.

#### **INPUT EAX = 16H: Returns Processor Frequency Information**

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 2-8.

### **METHODS FOR RETURNING BRANDING INFORMATION**

Use the following techniques to access branding information:

1. Processor brand string method; this method also returns the processor's maximum operating frequency
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

#### **The Processor Brand String Method**

Figure 2-7 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers.

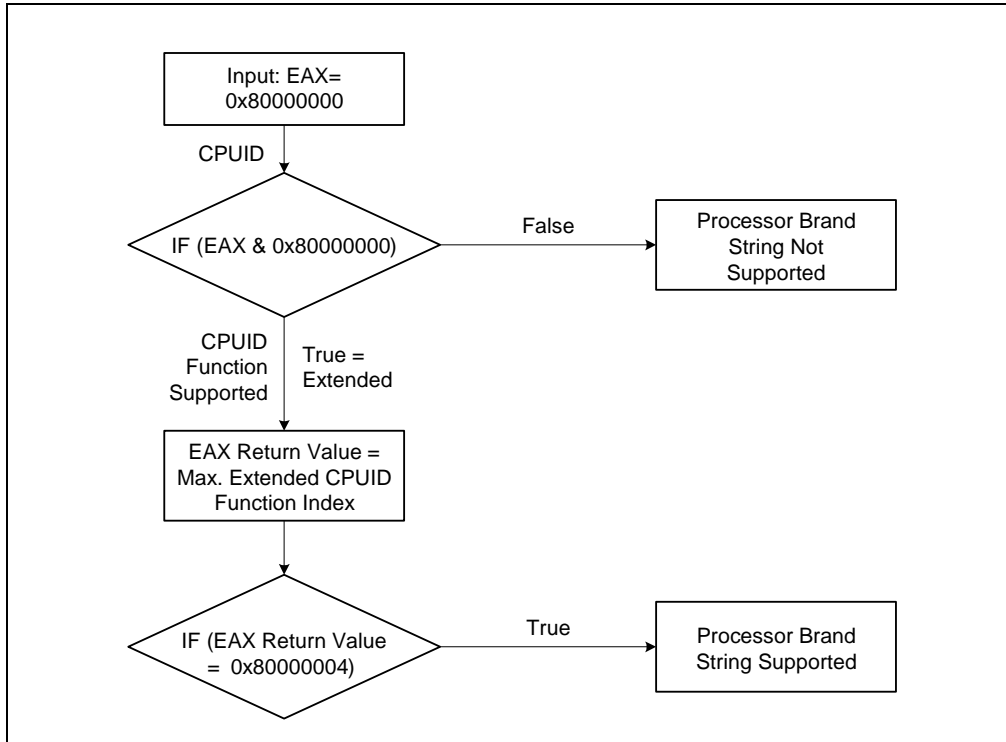


Figure 2-7. Determination of Support for the Processor Brand String

### How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 2-15 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 2-15. Processor Brand String Returned with Pentium 4 Processor

| EAX Input Value | Return Values  | ASCII Equivalent                    |
|-----------------|--|-------------------------------------|
| 80000002H       | EAX = 20202020H<br>EBX = 20202020H<br>ECX = 20202020H<br>EDX = 6E492020H | " "<br>" "<br>" "<br>"nl "          |
| 80000003H       | EAX = 286C6574H<br>EBX = 50202952H<br>ECX = 69746E65H<br>EDX = 52286D75H | "(let"<br>"P)R"<br>"itne"<br>"R(mu" |
| 80000004H       | EAX = 20342029H<br>EBX = 20555043H<br>ECX = 30303531H<br>EDX = 007A484DH | " 4)"<br>"UPC"<br>"0051"<br>"\0zHM" |

### Extracting the Maximum Processor Frequency from Brand Strings

Figure 2-8 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

#### NOTE

When a frequency is given in a brand string, it is the maximum qualified frequency of the processor, not the frequency at which the processor is currently running.

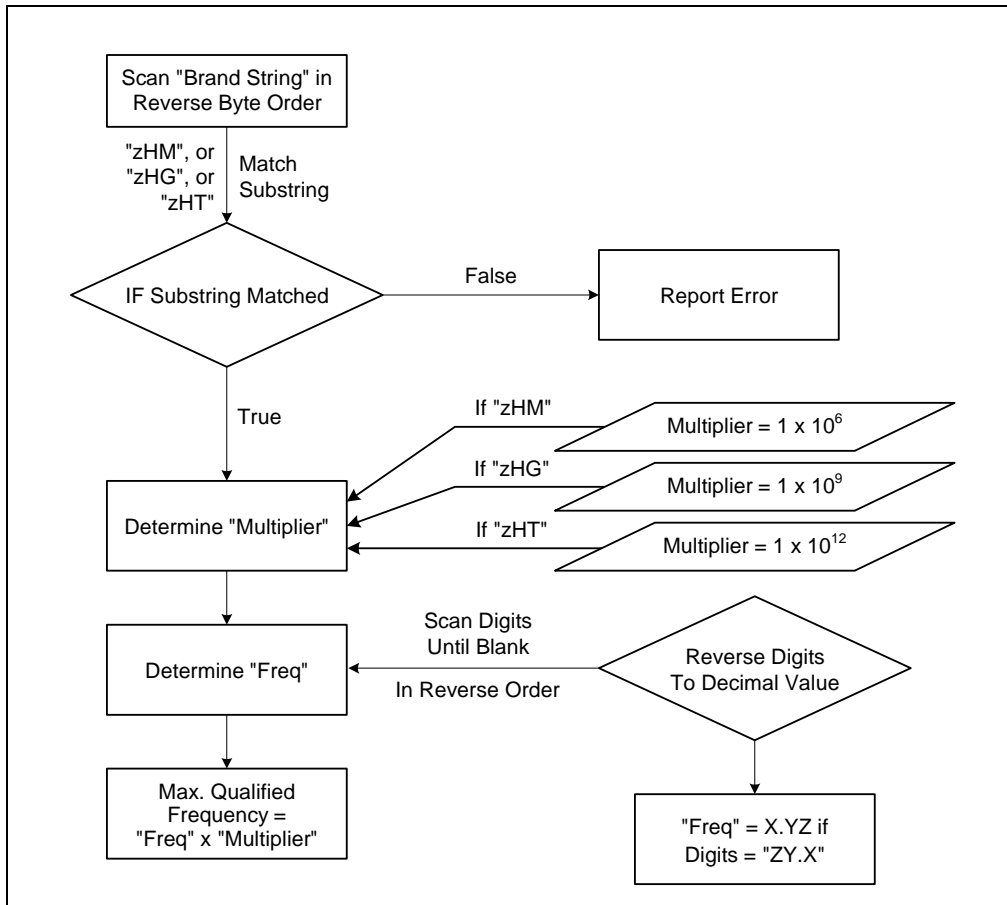


Figure 2-8. Algorithm for Extracting Maximum Processor Frequency

### The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 01H, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 2-16 shows brand indices that have identification strings associated with them.

**Table 2-16. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings**

| Brand Index | Brand String  |
|-------------|---|
| 00H         | This processor does not support the brand identification feature  |
| 01H         | Intel(R) Celeron(R) processor <sup>1</sup>  |
| 02H         | Intel(R) Pentium(R) III processor <sup>1</sup>  |
| 03H         | Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor |
| 04H         | Intel(R) Pentium(R) III processor   |
| 06H         | Mobile Intel(R) Pentium(R) III processor-M  |
| 07H         | Mobile Intel(R) Celeron(R) processor <sup>1</sup>   |
| 08H         | Intel(R) Pentium(R) 4 processor   |
| 09H         | Intel(R) Pentium(R) 4 processor   |
| 0AH         | Intel(R) Celeron(R) processor <sup>1</sup>  |
| 0BH         | Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP                |
| 0CH         | Intel(R) Xeon(R) processor MP   |
| 0EH         | Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor     |
| 0FH         | Mobile Intel(R) Celeron(R) processor <sup>1</sup>   |
| 11H         | Mobile Genuine Intel(R) processor   |
| 12H         | Intel(R) Celeron(R) M processor   |
| 13H         | Mobile Intel(R) Celeron(R) processor <sup>1</sup>   |
| 14H         | Intel(R) Celeron(R) processor   |
| 15H         | Mobile Genuine Intel(R) processor   |
| 16H         | Intel(R) Pentium(R) M processor   |
| 17H         | Mobile Intel(R) Celeron(R) processor <sup>1</sup>   |
| 18H - 0FFH  | RESERVED  |

## NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

**IA-32 Architecture Compatibility**

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

**Operation**

IA32\_BIOS\_SIGN\_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF

EAX = 0:

EAX ← Highest basic function input value understood by CPUID;

EBX ← Vendor identification string;

EDX ← Vendor identification string;

ECX ← Vendor identification string;

BREAK;

EAX = 1H:

EAX[3:0] ← Stepping ID;

EAX[7:4] ← Model;

EAX[11:8] ← Family;  
 EAX[13:12] ← Processor type;  
 EAX[15:14] ← Reserved;  
 EAX[19:16] ← Extended Model;  
 EAX[27:20] ← Extended Family;  
 EAX[31:28] ← Reserved;  
 EBX[7:0] ← Brand Index; (\* Reserved if the value is zero. \*)  
 EBX[15:8] ← CLFLUSH Line Size;  
 EBX[16:23] ← Reserved; (\* Number of threads enabled = 2 if MT enable fuse set. \*)  
 EBX[24:31] ← Initial APIC ID;  
 ECX ← Feature flags; (\* See Figure 2-5. \*)  
 EDX ← Feature flags; (\* See Figure 2-6. \*)

BREAK;

EAX = 2H:

EAX ← Cache and TLB information;  
 EBX ← Cache and TLB information;  
 ECX ← Cache and TLB information;  
 EDX ← Cache and TLB information;

BREAK;

EAX = 3H:

EAX ← Reserved;  
 EBX ← Reserved;  
 ECX ← ProcessorSerialNumber[31:0];  
 (\* Pentium III processors only, otherwise reserved. \*)  
 EDX ← ProcessorSerialNumber[63:32];  
 (\* Pentium III processors only, otherwise reserved. \*)

BREAK

EAX = 4H:

EAX ← Deterministic Cache Parameters Leaf; (\* See Table 2-8. \*)  
 EBX ← Deterministic Cache Parameters Leaf;  
 ECX ← Deterministic Cache Parameters Leaf;  
 EDX ← Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:

EAX ← MONITOR/MWAIT Leaf; (\* See Table 2-8. \*)  
 EBX ← MONITOR/MWAIT Leaf;  
 ECX ← MONITOR/MWAIT Leaf;  
 EDX ← MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:

EAX ← Thermal and Power Management Leaf; (\* See Table 2-8. \*)  
 EBX ← Thermal and Power Management Leaf;  
 ECX ← Thermal and Power Management Leaf;  
 EDX ← Thermal and Power Management Leaf;

BREAK;

EAX = 7H:

EAX ← Structured Extended Feature Leaf; (\* See Table 2-8. \*);  
 EBX ← Structured Extended Feature Leaf;  
 ECX ← Structured Extended Feature Leaf;  
 EDX ← Structured Extended Feature Leaf;

BREAK;

EAX = 8H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;



```

    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 2-8. *)
    EBX ← Direct Cache Access Information Leaf;
    ECX ← Direct Cache Access Information Leaf;
    EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 2-8. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
EAX = BH:
    EAX ← Extended Topology Enumeration Leaf; (* See Table 2-8. *)
    EBX ← Extended Topology Enumeration Leaf;
    ECX ← Extended Topology Enumeration Leaf;
    EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = DH:
    EAX ← Processor Extended State Enumeration Leaf; (* See Table 2-8. *)
    EBX ← Processor Extended State Enumeration Leaf;
    ECX ← Processor Extended State Enumeration Leaf;
    EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = 14H:
    EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 2-8. *)
    EBX ← Intel Processor Trace Enumeration Leaf;
    ECX ← Intel Processor Trace Enumeration Leaf;
    EDX ← Intel Processor Trace Enumeration Leaf;
BREAK;
EAX = 16H:
    EAX ← Processor Frequency Information Enumeration Leaf; (* See Table 2-8. *)
    EBX ← Processor Frequency Information Enumeration Leaf;
    ECX ← Processor Frequency Information Enumeration Leaf;
    EDX ← Processor Frequency Information Enumeration Leaf;
BREAK;
BREAK;
EAX = 80000000H:
    EAX ← Highest extended function input value understood by CPUID;
    EBX ← Reserved;
    ECX ← Reserved;
    EDX ← Reserved;
BREAK;
EAX = 80000001H:
    EAX ← Reserved;

```

EBX ← Reserved;  
 ECX ← Extended Feature Bits (\* See Table 2-8.\*);  
 EDX ← Extended Feature Bits (\* See Table 2-8.\*);

BREAK;

EAX = 80000002H:

EAX ← Processor Brand String;  
 EBX ← Processor Brand String, continued;  
 ECX ← Processor Brand String, continued;  
 EDX ← Processor Brand String, continued;

BREAK;

EAX = 80000003H:

EAX ← Processor Brand String, continued;  
 EBX ← Processor Brand String, continued;  
 ECX ← Processor Brand String, continued;  
 EDX ← Processor Brand String, continued;

BREAK;

EAX = 80000004H:

EAX ← Processor Brand String, continued;  
 EBX ← Processor Brand String, continued;  
 ECX ← Processor Brand String, continued;  
 EDX ← Processor Brand String, continued;

BREAK;

EAX = 80000005H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX = 80000006H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Cache information;  
 EDX ← Reserved = 0;

BREAK;

EAX = 80000007H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX = 80000008H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

DEFAULT: (\* EAX = Value outside of recognized range for CPUID. \*)

(\* If the highest basic information leaf data depend on ECX input value, ECX is honored.\*)

EAX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
 EBX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
 ECX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
 EDX ← Reserved; (\* Information returned for highest basic information leaf. \*)

BREAK;

ESAC;

**Flags Affected**

None.

**Exceptions (All Operating Modes)**

#UD                    If the LOCK prefix is used.

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.§

This page was  
intentionally left  
blank.

## CHAPTER 3

# SYSTEM PROGRAMMING FOR INTEL® AVX-512

---

This chapter describes the operating system programming considerations for supporting the following extended processor states: 512-bit ZMM registers and opmask k-registers. These system programming requirements apply to AVX-512 Foundation instructions and other 512-bit instructions described in Chapter 7.

The basic requirements for an operating system using XSAVE/XRSTOR to manage processor extended states, e.g. YMM registers, can be found in Chapter 13 of *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A*. This chapter covers additional requirements for OS to support ZMM and opmask register states.

### 3.1 AVX-512 STATE, EVEX PREFIX AND SUPPORTED OPERATING MODES

AVX-512 instructions are encoded using EVEX prefix. The EVEX encoding scheme can support 512-bit, 256-bit and 128-bit instructions that operate on opmask register, ZMM, YMM and XMM states.

For processors that support AVX-512 family of instructions, the extended processor states (ZMM and opmask registers) exist in all operating modes. However, the access to those states may vary in different modes. The processor's support for instruction extensions that employ EVEX prefix encoding is independent of the processor's support for using XSAVE/XRSTOR/XSAVEOPT to those states.

Instructions requiring EVEX prefix encoding generally are supported in 64-bit, 32-bit modes, and 16-bit protected mode. They are not supported in Real mode, Virtual-8086 mode or entering into SMM mode.

Note that bits MAX\_VL-1:256 (511:256) of ZMM register state are maintained across transitions into and out of these modes. Because the XSAVE/XRSTOR/XSAVEOPT instruction can operate in all operating modes, it is possible that the processor's ZMM register state can be modified by software in any operating mode by executing XRSTOR. The ZMM registers can be updated by XRSTOR using the state information stored in the XSAVE/XRSTOR area residing in memory.

### 3.2 AVX-512 STATE MANAGEMENT

Operating systems must use the XSAVE/XRSTOR/XSAVEOPT instructions for ZMM and opmask state management. An OS must enable its ZMM and opmask state management to support AVX-512 Foundation instructions. Otherwise, an attempt to execute an instruction in AVX-512 Foundation instructions (including a scalar 128-bit SIMD instructions using EVEX encoding) will cause a #UD exception. An operating system, which enabled AVX-512 state to support AVX-512 Foundation instructions, is also sufficient to support the rest of AVX-512 family of instructions.

#### 3.2.1 Detection of ZMM and Opmask State Support

Hardware support of the extended state components for executing AVX-512 Foundation instructions is queried through the main leaf of CPUID leaf function 0DH with index ECX = 0. Specifically, the return value in EDX:EAX of CPUID.(EAX=0DH, ECX=0) provides a 64-bit wide bit vector of hardware support of processor state components, beginning with bit 0 of EAX corresponding to x87 FPU state, CPUID.(EAX=0DH, ECX=0):EAX[1] corresponding to SSE state (XMM registers and MXCSR), CPUID.(EAX=0DH, ECX=0):EAX[2] corresponding to YMM states.

The ZMM and opmask states consist of three additional components in the XSAVE/XRSTOR state save area:

- The opmask register state component represents eight 64-bit opmask registers. Processor support for this component state is indicated by CPUID.(EAX=0DH, ECX=0):EAX[5].
- The ZMM\_Hi256 component represents the high 256 bits of the low 16 ZMM registers, i.e. ZMM0..15[511:256]. Processor support for this component state is indicated by CPUID.(EAX=0DH, ECX=0):EAX[6].
- The Hi16\_ZMM component represents the full 512 bits of the high 16 ZMM registers, i.e. ZMM16..31[511:0]. Processor support for this component state is indicated by CPUID.(EAX=0DH, ECX=0):EAX[7].

Each component state has a corresponding enable it in the XCR0 register. Operating system must use XSETBV to set these three enable bits to enable AVX-512 Foundation instructions register to be decoded. The location of bit vector representing the AVX-512 states, matching the layout of the XCR0 register, is provided in the following figure.

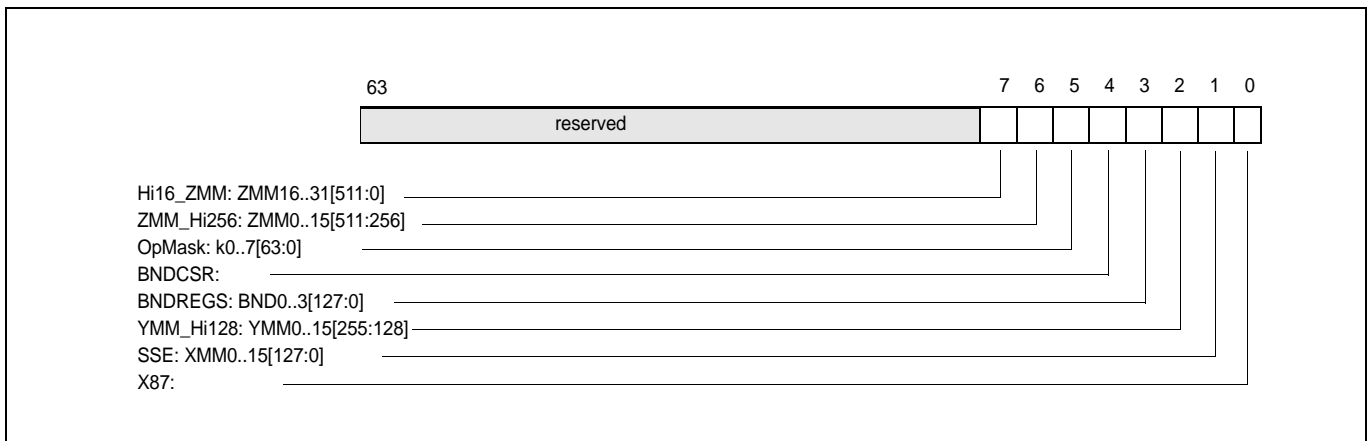


Figure 3-1. Bit Vector and XCR0 Layout of Extended Processor State Components

### 3.2.2 Enabling of ZMM and Opmask Register State

An OS can enable ZMM and opmask register state support with the following steps:

- Verify the processor supports XSAVE/XRSTOR/XSETBV/XGETBV instructions and the XCR0 register by checking CPUID.1.ECX.XSAVE[bit 26]=1.
- Verify the processor supports SSE, YMM, ZMM\_Hi256, Hi16\_ZMM, and opmask states (i.e. bits 2:1 and 7:5 of XCR0 are valid) by checking CPUID.(EAX=0DH, ECX=0):EAX[7:5].

The OS must determine the buffer size requirement for the XSAVE area that will be used by XSAVE/XRSTOR. Note that even though ZMM8-ZMM31 are not accessible in 32 bit mode, a 32 bit OS is still required to allocate the buffer for the entire ZMM state.

- Set CR4.OSXSAVE[bit 18]=1 to enable the use of XSETBV/XGETBV instructions to write/read the XCR0 register.
- Supply an appropriate mask via EDX:EAX to execute XSETBV to enable the processor state components that the OS wishes to manage using XSAVE/XRSTOR instruction.

To enable ZMM and opmask register state, system software must use a EDX:EAX mask of 111xx111b when executing XSETBV.

Table 3-1. XCR0 Processor State Components

| Bit           | Meaning  |
|---------------|--|
| 0 - x87       | This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.  |
| 1 - SSE       | If 1, the processor supports SSE state (MXCSR and XMM registers) management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.                                    |
| 2 - YMM_Hi128 | If 1, the processor supports YMM_hi128 state management (upper 128 bits of YMM0-15) using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.                            |
| 3 - BNDREGS   | If 1, the processor supports Intel Memory Protection Extensions (Intel MPX) bound register state management using XSAVE, XSAVEOPT, and XRSTOR. See Section 9.3.2 for system programming requirement to enable Intel MPX. |
| 4 - BNDCSR    | If 1, the processor supports Intel MPX bound configuration and status management using XSAVE, XSAVEOPT, and XRSTOR. See Section 9.3.2 for system programming requirement to enable Intel MPX.                            |

**Table 3-1. XCR0 Processor State Components**

| Bit           | Meaning  |
|---------------|--|
| 5 - Opmask    | If 1, the processor supports the opmask state management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.   |
| 6 - ZMM_Hi256 | If 1, the processor supports ZMM_Hi256 state (the upper 256 bits of the low 16 ZMM registers) management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions. |
| 7 - Hi16_ZMM  | If 1, the processor supports Hi16_ZMM state (the full 512 bits of the high16 ZMM registers) management using XSAVE, XSAVEOPT, and XRSTOR. This bit must be set to '1' to enable AVX-512 Foundation instructions.   |

### 3.2.3 Enabling of SIMD Floating-Exception Support

AVX-512 Foundation instructions may generate SIMD floating-point exceptions. An OS must enable SIMD floating-point exception support by setting CR4.OSXMMEXCPT[bit 10]=1.

The effect of CR4 setting that affects AVX-512 Foundation instructions is the same as for AVX and FMA enabling as listed in Table 3-2

**Table 3-2. CR4 Bits for AVX-512 Foundation Instructions Technology Support**

| Bit                    | Meaning   |
|------------------------|---|
| CR4.OSXSAVE[bit 18]    | If set, the OS supports use of XSETBV/XGETBV instruction to access the XCR0 register, XSAVE/XRSTOR to manage processor extended states. Must be set to '1' to enable AVX-512 Foundation, AVX2, FMA, and AVX instructions.           |
| CR4.OSXMMEXCPT[bit 10] | Must be set to 1 to enable SIMD floating-point exceptions. This applies to SIMD floating-point instructions across AVX-512 Foundation, AVX and FMA, and legacy 128-bit SIMD floating-point instructions operating on XMM registers. |
| CR4.OSFXSR[bit 9]      | Must be set to 1 to enable legacy 128-bit SIMD instructions operating on XMM state. Not needed to enable AVX-512 Foundation, AVX2, FMA, and AVX instructions.   |

### 3.2.4 The Layout of XSAVE State Save Area

The OS must determine the buffer size requirement by querying CPUID with EAX=0DH, ECX=0. If the OS wishes to enable all processor extended state components in the XCR0, it can allocate the buffer size according to CPUID.(EAX=0DH, ECX=0):ECX.

After the memory buffer for XSAVE is allocated, the entire buffer must be cleared prior to executing XSAVE.

The XSAVE area layout currently defined in Intel Architecture is listed in Table 3-3. The register fields of the first 512 byte of the XSAVE area are identical to those of the FXSAVE/FXRSTOR area.

The layout of the XSAVE Area for additional processor components (512-bit ZMM register, 32 ZMM registers, opmask registers) are to be determined later.

**Table 3-3. Layout of XSAVE Area For Processor Supporting YMM State**

| Save Areas                  | Offset (Byte)              | Size (Bytes)               |
|-----------------------------|----------------------------|----------------------------|
| FPU/SSE SaveArea            | 0                          | 512                        |
| Header                      | 512                        | 64                         |
| Ext_Save_Area_2 (YMM_Hi128) | CPUID.(EAX=0DH, ECX=2):EBX | CPUID.(EAX=0DH, ECX=2):EAX |
| Ext_Save_Area_3 (BNDREGS)   | CPUID.(EAX=0DH, ECX=3):EBX | CPUID.(EAX=0DH, ECX=3):EAX |
| Ext_Save_Area_4 (BNDCSR)    | CPUID.(EAX=0DH, ECX=4):EBX | CPUID.(EAX=0DH, ECX=4):EAX |
| Ext_Save_Area_5 (OPMASK)    | CPUID.(EAX=0DH, ECX=5):EBX | CPUID.(EAX=0DH, ECX=5):EAX |
| Ext_Save_Area_6 (ZMM_Hi256) | CPUID.(EAX=0DH, ECX=6):EBX | CPUID.(EAX=0DH, ECX=6):EAX |
| Ext_Save_Area_7 (Hi16_ZMM)  | CPUID.(EAX=0DH, ECX=7):EBX | CPUID.(EAX=0DH, ECX=7):EAX |

The format of the header is as follows (see Table 3-4):

**Table 3-4. XSAVE Header Format**

| 15:8                    | 7:0                     | Byte Offset from Header | Byte Offset from XSAVE Area |
|-------------------------|-------------------------|-------------------------|-----------------------------|
| Reserved (Must be zero) | XSTATE_BV               | 0                       | 512                         |
| Reserved                | Reserved (Must be zero) | 16                      | 528                         |
| Reserved                | Reserved                | 32                      | 544                         |
| Reserved                | Reserved                | 48                      | 560                         |

The layout of the Ext\_Save\_Area[YMM\_Hi128] contains 16 of the upper 128-bits of the YMM registers, it is shown in Table 3-5.

**Table 3-5. XSAVE Save Area Layout for YMM\_Hi128 State (Ext\_Save\_Area\_2)**

| 31 16          | 15 0           | Byte Offset from YMM_Hi128_Save_Area | Byte Offset from XSAVE Area |
|----------------|----------------|--------------------------------------|-----------------------------|
| YMM1[255:128]  | YMM0[255:128]  | 0                                    | 576                         |
| YMM3[255:128]  | YMM2[255:128]  | 32                                   | 608                         |
| YMM5[255:128]  | YMM4[255:128]  | 64                                   | 640                         |
| YMM7[255:128]  | YMM6[255:128]  | 96                                   | 672                         |
| YMM9[255:128]  | YMM8[255:128]  | 128                                  | 704                         |
| YMM11[255:128] | YMM10[255:128] | 160                                  | 736                         |
| YMM13[255:128] | YMM12[255:128] | 192                                  | 768                         |
| YMM15[255:128] | YMM14[255:128] | 224                                  | 800                         |

The layout of the Ext\_SAVE\_Area\_3[BNDREGS] contains bounds register state of the Intel Memory Protection Extensions (Intel MPX), which is described in Section 9.3.2.

The layout of the Ext\_SAVE\_Area\_4[BNDCSR] contains the processor state of bounds configuration and status of Intel MPX, which is described in Section 9.3.2.

The layout of the Ext\_SAVE\_Area\_5[Opmask] contains 8 64-bit mask register as shown in Table 3-6.



**Table 3-6. XSAVE Save Area Layout for Opmask Registers**

| 15 8     | 7 0      | Byte Offset from OPMASK_Save_Area | Byte Offset from XSAVE Area |
|----------|----------|-----------------------------------|-----------------------------|
| K1[63:0] | K0[63:0] | 0                                 | 1088                        |
| K3[63:0] | K2[63:0] | 16                                | 1104                        |
| K5[63:0] | K4[63:0] | 32                                | 1120                        |
| K7[63:0] | K6[63:0] | 48                                | 1136                        |

The layout of the Ext\_SAVE\_Area\_6[ZMM\_Hi256] is shown below in Table 3-7.

**Table 3-7. XSAVE Save Area Layout for ZMM State of the High 256 Bits of ZMM0-ZMM15 Registers**

| 63 32          | 31 0           | Byte Offset from ZMM_Hi256_Save_Area | Byte Offset from XSAVE Area |
|----------------|----------------|--------------------------------------|-----------------------------|
| ZMM1[511:256]  | ZMM0[511:256]  | 0                                    | 1152                        |
| ZMM3[511:256]  | ZMM2[511:256]  | 64                                   | 1216                        |
| ZMM5[511:256]  | ZMM4[511:256]  | 128                                  | 1280                        |
| ZMM7[511:256]  | ZMM6[511:256]  | 192                                  | 1344                        |
| ZMM9[511:256]  | ZMM8[511:256]  | 256                                  | 1408                        |
| ZMM11[511:256] | ZMM10[511:256] | 320                                  | 1472                        |
| ZMM13[511:256] | ZMM12[511:256] | 384                                  | 1536                        |
| ZMM15[511:256] | ZMM14[511:256] | 448                                  | 1600                        |

The layout of the Ext\_SAVE\_Area\_7[Hi16\_ZMM] corresponding to the upper new 16 ZMM registers is shown below in Table 3-8.

**Table 3-8. XSAVE Save Area Layout for ZMM State of ZMM16-ZMM31 Registers**

| 127 64       | 63 0         | Byte Offset from Hi16_ZMM_Save_Area | Byte Offset from XSAVE Area |
|--------------|--------------|-------------------------------------|-----------------------------|
| ZMM17[511:0] | ZMM16[511:0] | 0                                   | 1664                        |
| ZMM19[511:0] | ZMM18[511:0] | 128                                 | 1792                        |
| ZMM21[511:0] | ZMM20[511:0] | 256                                 | 1920                        |
| ZMM23[511:0] | ZMM22[511:0] | 384                                 | 2048                        |
| ZMM25[511:0] | ZMM24[511:0] | 512                                 | 2176                        |
| ZMM27[511:0] | ZMM26[511:0] | 640                                 | 2304                        |
| ZMM29[511:0] | ZMM28[511:0] | 768                                 | 2432                        |
| ZMM31[511:0] | ZMM30[511:0] | 896                                 | 2560                        |

### 3.2.5 XSAVE/XRSTOR Interaction with YMM State and MXCSR

The processor's actions as a result of executing XRSTOR, on the MXCSR, XMM and YMM registers, are listed in Table 3-9. The XMM registers may be initialized by the processor (See XRSTOR operation in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*). When the MXCSR register is updated from memory, reserved bit checking is enforced. XSAVE / XRSTOR will save / restore the MXCSR only if the AVX or SSE bits are set in the EDX:EAX mask.

**Table 3-9. XRSTOR Action on MXCSR, XMM Registers, YMM Registers**

| EDX:EAX |       | XSTATE_BV |       | MXCSR      | YMM_Hi128 Registers | XMM Registers     |
|---------|-------|-----------|-------|------------|---------------------|-------------------|
| Bit 2   | Bit 1 | Bit 2     | Bit 1 |            |                     |                   |
| 0       | 0     | X         | X     | None       | None                | None              |
| 0       | 1     | X         | 0     | Load/Check | None                | Init by processor |
| 0       | 1     | X         | 1     | Load/Check | None                | Load              |
| 1       | 0     | 0         | X     | Load/Check | Init by processor   | None              |
| 1       | 0     | 1         | X     | Load/Check | Load                | None              |
| 1       | 1     | 0         | 0     | Load/Check | Init by processor   | Init by processor |
| 1       | 1     | 0         | 1     | Load/Check | Init by processor   | Load              |
| 1       | 1     | 1         | 0     | Load/Check | Load                | Init by processor |
| 1       | 1     | 1         | 1     | Load/Check | Load                | Load              |

The action of XSAVE for managing YMM and MXCSR is listed in Table 3-10.

**Table 3-10. XSAVE Action on MXCSR, XMM, YMM Register**

| EDX:EAX |       | XCR0_MASK |       | MXCSR | YMM_H Registers | XMM Registers |
|---------|-------|-----------|-------|-------|-----------------|---------------|
| Bit 2   | Bit 1 | Bit 2     | Bit 1 |       |                 |               |
| 0       | 0     | X         | X     | None  | None            | None          |
| 0       | 1     | X         | 1     | Store | None            | Store         |
| 0       | 1     | X         | 0     | None  | None            | None          |
| 1       | 0     | 0         | X     | None  | None            | None          |
| 1       | 0     | 1         | 1     | Store | Store           | None          |
| 1       | 1     | 0         | 0     | None  | None            | None          |
| 1       | 1     | 0         | 1     | Store | None            | Store         |
| 1       | 1     | 1         | 1     | Store | Store           | Store         |

### 3.2.6 XSAVE/XRSTOR/XSAVEOPT and Managing ZMM and Opmask States

The requirements for managing ZMM\_Hi256, Hi16\_ZMM and Opmask registers using XSAVE/XRSTOR/XSAVEOPT are simpler than those listed in Section 3.2.5. Because each of the three components (ZMM\_Hi256, Hi16\_ZMM and Opmask registers) can be managed independently of one another by XSAVE/XRSTOR/XSAVEOPT according to the corresponding bits in the bit vectors: EDX:EAX, XSAVE\_BV, XCR0\_MASK, independent of MXCSR:

- For using XSAVE with Opmask/ZMM\_Hi256/Hi16\_ZMM, XSAVE/XSAVEOPT will save the component to memory and mark the corresponding bits in the XSTATE\_BV of the XSAVE header, if that component is specified in EDX:EAX as input to XSAVE/XSAVEOPT.
- XRSTOR will restore the Opmask/ZMM\_Hi256/Hi16\_ZMM components by checking the corresponding bits in both the input bit vector in EDX:EAX of XRSTOR and in XSTATE\_BV of the header area in the following ways:
  - If the corresponding bit in EDX:EAX is set and XSTATE\_BV is INIT, that component will be initialized,
  - If the corresponding bit in EDX:EAX is set and XSTATE\_BV is set, that component will be restored from memory,
  - If the corresponding bit in EDX:EAX is not set, that component will remain unchanged.
- To enable AVX-512 Foundation instructions, all three components (Opmask/ZMM\_Hi256/Hi16\_ZMM) in XCR0 must be set.

The processor supplied INIT values for each processor state component used by XRSTOR is listed in Table 3-11.

**Table 3-11. Processor Supplied Init Values XRSTOR May Use**

| Processor State Component    | Processor Supplied Register Values   |
|------------------------------|--|
| x87 FPU State                | FCW ← 037FH; FTW ← 0FFFFH; FSW ← 0H; FPU CS ← 0H; FPU DS ← 0H; FPU IP ← 0H; FPU DP ← 0; ST0-ST7 ← 0; |
| SSE State <sup>1</sup>       | If 64-bit Mode: XMM0-XMM15 ← 0H;<br>Else XMM0-XMM7 ← 0H  |
| YMM_Hi128 State <sup>1</sup> | If 64-bit Mode: YMM0_H-YMM15_H ← 0H;<br>Else YMM0_H-YMM7_H ← 0H                                      |
| OPMASK State <sup>1</sup>    | If 64-bit Mode: K0-K7 ← 0H;  |
| ZMM_Hi256 State <sup>1</sup> | If 64-bit Mode: ZMM0_H-ZMM15_H ← 0H;<br>Else ZMM0_H-ZMM7_H ← 0H                                      |
| Hi16_ZMM State <sup>1</sup>  | If 64-bit Mode: ZMM16-ZMM31 ← 0H;  |

**NOTES:**

1. MXCSR state is not updated by processor supplied values. MXCSR state can only be updated by XRSTOR from state information stored in XSAVE/XRSTOR area.

### 3.3 RESET BEHAVIOR

At processor reset

- YMM0-15 bits[255:0] are set to zero.
- ZMM0-15 bits [511:256] are set to zero.
- ZMM16-31 are set to zero.
- Opmask register K0-7 are set to 0x0H.
- **XCRO**[2:1] is set to zero, **XCRO**[0] is set to 1.
- **XCRO**[7:6] and is set to zero, **XCRO**[Opmask] is set to 0.
- CR4.OSXSAVE[bit 18] (and its mirror CPUID.1.ECX.OSXSAVE[bit 27]) is set to 0.

### 3.4 EMULATION

Setting the CR0.EM bit to 1 provides a technique to emulate Legacy SSE floating-point instruction sets in software. This technique is not supported with AVX instructions, nor FMA instructions.

If an operating system wishes to emulate AVX instructions, set **XCRO**[2:1] to zero. This will cause AVX instructions to #UD. Emulation of FMA by operating system can be done similarly as with emulating AVX instructions.

### 3.5 WRITING FLOATING-POINT EXCEPTION HANDLERS

AVX-512, AVX and FMA floating-point exceptions are handled in an entirely analogous way to Legacy SSE floating-point exceptions. To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. The section titled "SSE and SSE2 SIMD Floating-Point Exceptions" in Chapter 11, "Programming with Streaming SIMD Extensions 2 (SSE2)," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (**#XM**), the CR4.OSXM-MEXCPT flag (bit 10) must be set.

This page was  
intentionally left  
blank.

## CHAPTER 4

# AVX-512 INSTRUCTION ENCODING

### 4.1 OVERVIEW SECTION

This chapter describes the details of AVX-512 instruction encoding system. The AVX-512 Foundation instruction described in Chapter 5 use a new prefix (called EVEX). Opmask instructions described in Chapter 6 are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix. The EVEX encoding architecture also applies to other 512-bit instructions described in Chapter 7.

The significant feature differences between EVEX and VEX are summarized below.

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions; opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g. packed instruction with “load+op” semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support “suppress all exceptions” functionality).

### 4.2 INSTRUCTION FORMAT AND EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 4-1:

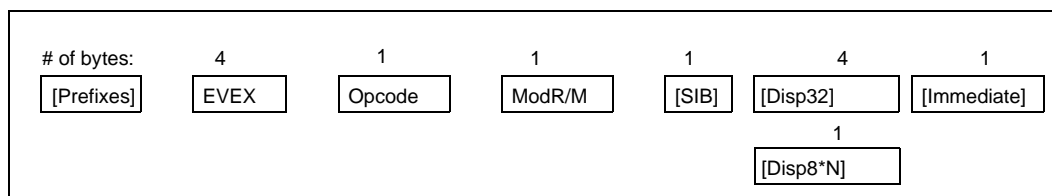


Figure 4-1. AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 4-2. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 4-2).

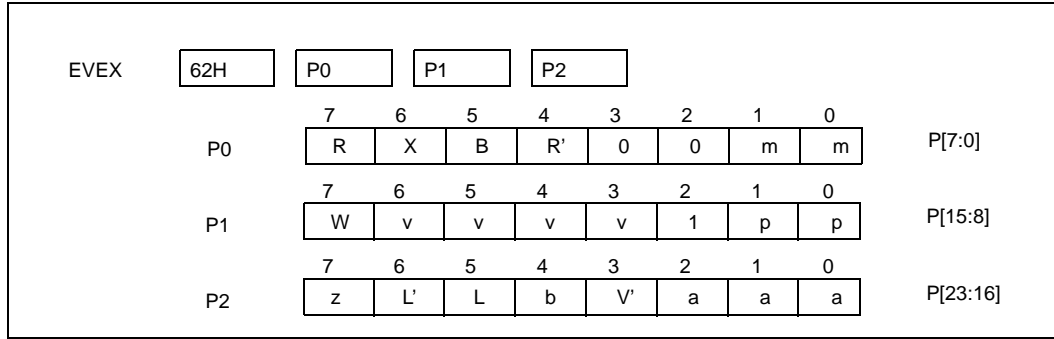


Figure 4-2. Bit Field Layout of the EVEX Prefix

Table 4-1. EVEX Prefix Bit Field Functional Grouping

| Notation  | Bit field Group                     | Position   | Comment   |
|-----------|-------------------------------------|------------|---|
| --        | Reserved                            | P[3 : 2]   | Must be 0   |
| --        | Fixed Value                         | P[10]      | Must be 1   |
| EVEX.mm   | Compressed legacy escape            | P[1 : 0]   | Identical to low two bits of VEX.mmmmm                  |
| EVEX.pp   | Compressed legacy prefix            | P[9 : 8]   | Identical to VEX.pp                                     |
| EVEX.RXB  | Next-8 register specifier modifier  | P[7 : 5]   | Combine with ModR/M.reg, ModR/M.rm (base, index/vidx)   |
| EVEXR'    | High-16 register specifier modifier | P[4]       | Combine with EVEX.R and ModR/M.reg                      |
| EVEXX     | High-16 register specifier modifier | P[6]       | Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent |
| EVEX.vvvv | NDS register specifier              | P[14 : 11] | Same as VEX.vvvv  |
| EVEXV'    | High-16 NDS/VIDX register specifier | P[19]      | Combine with EVEX.vvvv or when VSIB present             |
| EVEX.aaa  | Embedded opmask register specifier  | P[18 : 16] |   |
| EVEX.w    | Osize promotion/Opcode extension    | P[15]      |   |
| EVEX.z    | Zeroing/Merging                     | P[23]      |   |
| EVEX.b    | Broadcast/RC/SAE Context            | P[20]      |   |
| EVEX.L'L  | Vector length/RC                    | P[22 : 21] |   |

The bit fields in P[23:0] are divided into the following functional groups (Table 4-1 provides a tabular summary):

- Reserved bits: P[3:2] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.

- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
  - Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).
  - Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
  - For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.
- Vector length/rounding control specifier: P[22:21] can server one of three functionality:
  - vector length information for packed vector instructions,
  - ignored for instructions operating on vector register content as a single data element,
  - rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

### 4.3 REGISTER SPECIFIER ENCODING AND EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 4-2. Opmask register encoding is described in Section 4.3.1.

**Table 4-2. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits**

|                | 4 <sup>1</sup> | 3         | [2:0]     | Reg. Type   | Common Usages             |
|----------------|----------------|-----------|-----------|-------------|---------------------------|
| <b>REG</b>     | EVEX.R'        | REX.R     | modrm.reg | GPR, Vector | Destination or Source     |
| <b>NDS/NDD</b> | EVEX.V'        | EVEX.vvvv |           | GPR, Vector | 2ndSource or Destination  |
| <b>RM</b>      | EVEX.X         | EVEX.B    | modrm.r/m | GPR, Vector | 1st Source or Destination |
| <b>BASE</b>    | 0              | EVEX.B    | modrm.r/m | GPR         | memory addressing         |
| <b>INDEX</b>   | 0              | EVEX.X    | sib.index | GPR         | memory addressing         |
| <b>VIDX</b>    | EVEX.V'        | EVEX.X    | sib.index | Vector      | VSIB memory addressing    |
| <b>IS4</b>     | Imm8[3]        | Imm8[7:4] |           | Vector      | 3rd Source                |

**NOTES:**

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 4-3.

**Table 4-3. EVEX Encoding Register Specifiers in 32-bit Mode**

|            | [2:0]     | Reg. Type   | Common Usages  |
|------------|-----------|-------------|----------------|
| <b>REG</b> | modrm.reg | GPR, Vector | Dest or Source |

**Table 4-3. EVEX Encoding Register Specifiers in 32-bit Mode**

|                |           |             |                        |
|----------------|-----------|-------------|------------------------|
| <b>NDS/NDD</b> | EVEX.vvv  | GPR, Vector | 2ndSource or Dest      |
| <b>RM</b>      | modrm.r/m | GPR, Vector | 1st Source or Dest     |
| <b>BASE</b>    | modrm.r/m | GPR         | memory addressing      |
| <b>INDEX</b>   | sib.index | GPR         | memory addressing      |
| <b>VIDX</b>    | sib.index | Vector      | VSIB memory addressing |
| <b>IS4</b>     | Imm8[7:5] | Vector      | 3rd Source             |

### 4.3.1 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.
- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 4.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

**Table 4-4. Opmask Register Specifier Encoding**

|             | [2:0]     | Register Access     | Common Usages |
|-------------|-----------|---------------------|---------------|
| <b>REG</b>  | modrm.reg | k0-k7               | Source        |
| <b>NDS</b>  | VEX.vvv   | k0-k7               | 2ndSource     |
| <b>RM</b>   | modrm.r/m | k0-7                | 1st Source    |
| <b>{k1}</b> | EVEX.aaa  | k0 <sup>1</sup> -k7 | Opmask        |

#### NOTES:

1. instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

## 4.4 MASKING SUPPORT IN EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided in three different groups:

- Instructions which support "zeroing-masking".
  - Also allow merging-masking.



- Instructions which require  $aaa = 000$ .
  - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking
  - Require EVEX.z to be set to 0
  - This group is mostly composed of instructions that write to memory.
- Instructions which require  $aaa \neq 000$  do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

## 4.5 COMPRESSED DISPLACEMENT (DISP8\*N) SUPPORT IN EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 4-5 and Table 4-6 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 4-5 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 4.7).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less than dword do not support broadcasting using EVEX.b. These are listed in Table 4-6. Table 4-6 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 4-6. Instruction classified in Table 4-6 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype abbreviation will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8\*N rules still apply when using 16b addressing.

**Table 4-5. Compressed Displacement (DISP8\*N) Affected by Embedded Broadcast**

| TupleType        | EVEX.b | InputSize | EVEX.W | Broadcast | N (VL=128) | N (VL=256) | N (VL= 512) | Comment                           |
|------------------|--------|-----------|--------|-----------|------------|------------|-------------|-----------------------------------|
| Full Vector (FV) | 0      | 32bit     | 0      | none      | 16         | 32         | 64          | Load+Op (Full Vector Dword/Qword) |
|                  | 1      | 32bit     | 0      | {1tox}    | 4          | 4          | 4           |                                   |
|                  | 0      | 64bit     | 1      | none      | 16         | 32         | 64          |                                   |
|                  | 1      | 64bit     | 1      | {1tox}    | 8          | 8          | 8           |                                   |
| Half Vector (HV) | 0      | 32bit     | 0      | none      | 8          | 16         | 32          | Load+Op (Half Vector)             |
|                  | 1      | 32bit     | 0      | {1tox}    | 4          | 4          | 4           |                                   |

**Table 4-6. EVEX DISP8\*N For Instructions Not Affected by Embedded Broadcast**

| TupleType             | InputSize | EVEX.W | N (VL= 128) | N (VL= 256) | N (VL= 512) | Comment                            |
|-----------------------|-----------|--------|-------------|-------------|-------------|------------------------------------|
| Full Vector Mem (FVM) | N/A       | N/A    | 16          | 32          | 64          | Load/store or subDword full vector |

**Table 4-6. EVEX DISP8\*N For Instructions Not Affected by Embedded Broadcast(Continued)**

| TupleType           | InputSize | EVEX.W | N (VL= 128) | N (VL= 256) | N (VL= 512) | Comment                                |
|---------------------|-----------|--------|-------------|-------------|-------------|--|
| Tuple1 Scalar (T1S) | 8bit      | N/A    | 1           | 1           | 1           | 1 Tuple less than Full Vector          |
|                     | 16bit     | N/A    | 2           | 2           | 2           |  |
|                     | 32bit     | 0      | 4           | 4           | 4           |  |
|                     | 64bit     | 1      | 8           | 8           | 8           |  |
| Tuple1 Fixed (T1F)  | 32bit     | N/A    | 4           | 4           | 4           | 1 Tuple memsize not affected by EVEX.W |
|                     | 64bit     | N/A    | 8           | 8           | 8           |  |
| Tuple2 (T2)         | 32bit     | 0      | 8           | 8           | 8           | Broadcast (2 elements)                 |
|                     | 64bit     | 1      | NA          | 16          | 16          |  |
| Tuple4 (T4)         | 32bit     | 0      | NA          | 16          | 16          | Broadcast (4 elements)                 |
|                     | 64bit     | 1      | NA          | NA          | 32          |  |
| Tuple8 (T8)         | 32bit     | 0      | NA          | NA          | 32          | Broadcast (8 elements)                 |
| Half Mem (HVM)      | N/A       | N/A    | 8           | 16          | 32          | SubQword Conversion                    |
| QuarterMem (QVM)    | N/A       | N/A    | 4           | 8           | 16          | SubDword Conversion                    |
| OctMem (OVM)        | N/A       | N/A    | 2           | 4           | 8           | SubWord Conversion                     |
| Mem128 (M128)       | N/A       | N/A    | 16          | 16          | 16          | Shift count from memory                |
| MOVDDUP (DUP)       | N/A       | N/A    | 8           | 32          | 64          | VMOVDDUP                               |

## 4.6 EVEX ENCODING OF BROADCAST/ROUNDING/SAE SUPPORT

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with “load+op” semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- “Suppress All exceptions” (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

### 4.6.1 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single-precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

### 4.6.2 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L'L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, “suppress all exceptions” is implied. The processor behave as if all MXCSR masking controls are set.

### 4.6.3 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

### 4.6.4 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L'L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L'L bits. When EVEX is used to encode scalar instructions, L'L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 4-7.

**Table 4-7. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions**

| Position  | P2[4]  | P2[6:5]   | P2[6:5]   |
|---|--|---|---|
| Broadcast/Rounding/SAE Context                                    | EVEX.b                                       | EVEX.L'L  | EVEX.RC   |
| Reg-reg, FP Instructions w/ rounding semantic                     | Enable static rounding control (SAE implied) | Vector length Implied (512 bit or scalar)                           | 00b: SAE + RNE<br>01b: SAE + RD<br>10b: SAE + RU<br>11b: SAE + RZ |
| FP Instructions w/o rounding semantic, can cause #XF              | SAE control                                  | 00b: 128-bit<br>01b: 256-bit<br>10b: 512-bit<br>11b: Reserved (#UD) | NA  |
| Load+op Instructions w/ memory source                             | Broadcast Control                            |   | NA  |
| Other Instructions (Explicit Load/Store/Broadcast/Gather/Scatter) | Must be 0 (otherwise #UD)                    |   | NA  |

## 4.7 #UD EQUATIONS FOR EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

### 4.7.1 State Dependent #UD

In general, attempts to execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 4-8 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCRO shown in Table 4-8 will cause #UD.

**Table 4-8. OS XSAVE Enabling Requirements of Instruction Categories**

| Instruction Categories                            | Vector Register State Access | Required XCRO Bit Vector [7:0] |
|---|------------------------------|--------------------------------|
| Legacy SIMD prefix encoded Instructions (e.g SSE) | XMM                          | xxxxxx11b                      |
| VEV-encoded instructions operating on YMM         | YMM                          | xxxxx111b                      |
| EVEX-encoded 128-bit instructions                 | ZMM                          | 111xx111b                      |
| EVEX-encoded 256-bit instructions                 | ZMM                          | 111xx111b                      |

**Table 4-8. OS XSAVE Enabling Requirements of Instruction Categories**

|   |       |           |
|---|-------|-----------|
| <b>EVEX-encoded 512-bit instructions</b>            | ZMM   | 111xx111b |
| <b>VEX-encoded instructions operating on opmask</b> | k-reg | xx1xxx11b |

### 4.7.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 4-9:

**Table 4-9. Opcode Independent, State Dependent EVEX Bit Fields**

| Position | Notation | 64-bit #UD   | Non-64-bit #UD                 |
|----------|----------|--------------|--------------------------------|
| P[3 : 2] | --       | if > 0       | if > 0                         |
| P[10]    | --       | if 0         | if 0                           |
| P[1: 0]  | EVEX.mm  | if 00b       | if 00b                         |
| P[7 : 6] | EVEX.RX  | None (valid) | None (BOUND if EVEX.RX != 11b) |

### 4.7.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 4-10 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

**Table 4-10. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields**

| Notation  | Position   | Operand Encoding                      | 64-bit #UD     | Non-64-bit #UD                 |
|-----------|------------|---------------------------------------|----------------|--------------------------------|
| EVEX.R    | P[7]       | ModRM.reg encodes k-reg               | if EVEX.R = 0  | None (BOUND if EVEX.RX != 11b) |
|           |            | ModRM.reg is opcode extension         | None (ignored) |                                |
|           |            | ModRM.reg encodes all other registers | None (valid)   |                                |
| EVEX.X    | P[6]       | ModRM.r/m encodes ZMM/YMM/XMM         | None (valid)   |                                |
|           |            | ModRM.r/m encodes k-reg or GPR        | None (ignored) |                                |
|           |            | ModRM.r/m without SIB/VSIB            | None (ignored) |                                |
|           |            | ModRM.r/m with SIB/VSIB               | None (valid)   |                                |
| EVEX.B    | P[5]       | ModRM.r/m encodes k-reg               | None (ignored) | None (ignored)                 |
|           |            | ModRM.r/m encodes other registers     | None (valid)   |                                |
|           |            | ModRM.r/m base present                | None (valid)   |                                |
|           |            | ModRM.r/m base not present            | None (ignored) |                                |
| EVEXR'    | P[4]       | ModRM.reg encodes k-reg or GPR        | if 0           | None (ignored)                 |
|           |            | ModRM.reg is opcode extension         | None (ignored) |                                |
|           |            | ModRM.reg encodes ZMM/YMM/XMM         | None (valid)   |                                |
| EVEX.vvvv | P[14 : 11] | vvvv encodes ZMM/YMM/XMM              | None (valid)   | None (valid)<br>P[14] ignored  |
|           |            | otherwise                             | if != 1111b    | if != 1111b                    |
| EVEXV'    | P[19]      | encodes ZMM/YMM/XMM                   | None (valid)   | if 0                           |
|           |            | otherwise                             | if 0           | if 0                           |

Table 4-11 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEZ.z

**Table 4-11. #UD Conditions of Opmask Related Encoding Field**

| Notation | Position   | Operand Encoding  | 64-bit #UD                 | Non-64-bit #UD             |
|----------|------------|---|----------------------------|----------------------------|
| EVEX.aaa | P[18 : 16] | instructions do not use opmask for conditional processing <sup>1</sup>            | if aaa != 000b             | if aaa != 000b             |
|          |            | opmask used as conditional processing mask and updated at completion <sup>2</sup> | if aaa = 000b              | if aaa = 000b;             |
|          |            | opmask used as conditional processing   | None (valid <sup>3</sup> ) | None (valid <sup>1</sup> ) |
| EVEZ.z   | P[23]      | vector instruction using opmask as source or destination <sup>4</sup>             | if EVEZ.z != 0             | if EVEZ.z != 0             |
|          |            | store instructions or gather/scatter instructions                                 | if EVEZ.z != 0             | if EVEZ.z != 0             |
|          |            | instruction supporting conditional processing mask with EVEX.aaa = 000b           | if EVEZ.z != 0             | if EVEZ.z != 0             |

**NOTES:**

1. E.g. VBROADCASTMxxx, VPMOVM2x, VPMOVx2M
2. E.g. Gather/Scatter family
3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in K0.
4. E.g. VFPCCLASSPD/PS, VCMQB/D/Q/W family, VPMOVM2x, VPMOVx2M

Table 4-12 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

**Table 4-12. #UD Conditions Dependent on EVEX.b Context**

| Notation  | Position   | Operand Encoding                                    | 64-bit #UD                 | Non-64-bit #UD             |
|-----------|------------|---|----------------------------|----------------------------|
| EVEX.L'Lb | P[22 : 20] | reg-reg, FP instructions with rounding semantic     | None (valid <sup>1</sup> ) | None (valid <sup>1</sup> ) |
|           |            | other reg-reg, FP instructions that can cause #XF   | None (valid <sup>2</sup> ) | None (valid <sup>2</sup> ) |
|           |            | other reg-mem instructions in Table 4-5             | None (valid <sup>3</sup> ) | None (valid <sup>3</sup> ) |
|           |            | other instruction classes <sup>4</sup> in Table 4-6 | if EVEX.b > 0              | if EVEX.b > 0              |

**NOTES:**

1. L'L specifies rounding control, see Table 4-7, supports {er} syntax.
2. L'L specifies vector length, see Table 4-7, supports {sae} syntax.
3. L'L specifies vector length, see Table 4-7, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

## 4.8 DEVICE NOT AVAILABLE

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

## 4.9 SCALAR INSTRUCTIONS

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

## 4.10 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of “E##” or with a suffix “E##XX”. The “##” designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with “Load+op” semantic supports memory fault suppression, which is represented by E##. The instructions with “Load+op” semantic but do not support fault suppression are named “E##NF”. A summary table of exception classes by class names are shown below.

**Table 4-13. EVEX-Encoded Instruction Exception Class Summary**

| Exception Class | Instruction set                    | Mem arg                                      | (#XM) |
|-----------------|------------------------------------|--|-------|
| Type E1         | Vector Moves/Load/Stores           | explicitly aligned, w/ fault suppression     | none  |
| Type E1NF       | Vector Non-temporal Stores         | explicitly aligned, no fault suppression     | none  |
| Type E2         | FP Vector Load+op                  | Support fault suppression                    | yes   |
| Type E2NF       | FP Vector Load+op                  | No fault suppression                         | yes   |
| Type E3         | FP Scalar/Partial Vector, Load+Op  | Support fault suppression                    | yes   |
| Type E3NF       | FP Scalar/Partial Vector, Load+Op  | No fault suppression                         | yes   |
| Type E4         | Integer Vector Load+op             | Support fault suppression                    | no    |
| Type E4NF       | Integer Vector Load+op             | No fault suppression                         | no    |
| Type E5         | Legacy-like Promotion              | Varies, Support fault suppression            | no    |
| Type E5NF       | Legacy-like Promotion              | Varies, No fault suppression                 | no    |
| Type E6         | Post AVX Promotion                 | Varies, w/ fault suppression                 | no    |
| Type E6NF       | Post AVX Promotion                 | Varies, no fault suppression                 | no    |
| Type E7NM       | register-to-register op            | none   | none  |
| Type E9NF       | Miscellaneous 128-bit              | Vector-length Specific, no fault suppression | none  |
| Type E10        | Non-XF Scalar                      | Vector Length ignored, w/ fault suppression  | none  |
| Type E10NF      | Non-XF Scalar                      | Vector Length ignored, no fault suppression  | none  |
| Type E11        | VCVTPH2PS                          | Half Vector Length, w/ fault suppression     | yes   |
| Type E11NF      | VCVTPS2PH                          | Half Vector Length, no fault suppression     | yes   |
| Type E12        | Gather and Scatter Family          | VSIB addressing, w/ fault suppression        | none  |
| Type E12NP      | Gather and Scatter Prefetch Family | VSIB addressing, w/o page fault              | none  |

Table 4-14 lists EVEX-encoded instruction mnemonic by exception classes.

**Table 4-14. EVEX Instructions in each Exception Class**

| Exception Class | Instruction                             |
|-----------------|---|
| Type E1         | VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64  |
| Type E1NF       | VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS |

Table 4-14. EVEX Instructions in each Exception Class(Continued)

| Exception Class      | Instruction  |
|----------------------|--|
| Type E2              | VADDPD, VADDPs, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPS2DQ, VCVTTPD2DQ, VCVTTPS2DQ, VDIVPD, VDIVPS, VFMADDxxxPD, VFMADDxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS   |
|                      | VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2UDQS, VCVTQQ2PD, VCVTQQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VFIXUPIMMPD, VFIXUPIMMPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VSCALEFPD, VSCALEFPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS  |
| Type E3              | VADDSd, VADDSs, VCMPSD, VCMPSs, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VDIVSD, VDIVSS, VMAXSD, VMAXSS, VMINSd, VMINSs, VMULSD, VMULSS, VSQRTSD, VSQRTSS, VSUBSD, VSUBSS  |
|                      | VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, VCVTTPS2UQQ, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSs, VGETMANTSD, VGETMANTSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS   |
| Type E3NF            | VCOMISD, VCOMISS, VCVTSD2SI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTSS2SI, VUCOMISD, VUCOMISS  |
|                      | VCVTSD2USI, VCVTSS2USI, VCVTSS2USI, VCVTSS2USI, VCVTUSI2SD, VCVTUSI2SS   |
| Type E4              | VANDPD, VANDPS, VANDNPD, VANDNPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDd, VPADDQ, VPANDd, VPANDQ, VPANDND, VPANDNQ, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSd, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VXORPD, VXORPS, VPSLLVD, VPSLLVQ,   |
|                      | VBLENDMPD, VBLENDMPS, VPBLENDMD, VPBLENDMQ, VFPCCLASSPD, VFPCCLASSPS, VPCMPD, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPROLD, VPROLQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ) <sup>1</sup> , VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VPCONFLICTD, VPCONFLICTQ, VPSRAVw, VPSRAVD, VPSRAVw, VPSRAVQ, VPMADD52LUQ, VPMADD52HUQ   |
| E4.nb <sup>2</sup>   | VMOVUPD, VMOVUPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VPCMPB, VPCMPW, VPCMPUB, VPCMPUW, VEXPANDPD, VEXPANDPS, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VCOMPRESSPD, VCOMPRESSPS, VPABSB, VPABSw, VPADDB, VPADDW, VPADDSB, VPADDSw, VPADDUSB, VPADDUSw, VPAVGB, VPAVGW, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPMAXSB, VPMAXSw, VPMAXUB, VPMAXUW, VPMINSB, VPMINSw, VPMINUB, VPMINUw, VPMULHRSw, VPMULHUw, VPMULHW, VPMULLw, VPSUBB, VPSUBW, VPSUBSB, VPSUBSw, VPTESTMB, VPTESTMw, VPTESTNMB, VPTESTNMw, VPSLLW, VPSRAW, VPSRLW, VPSLLWw, VPSRLWw |
| Type E4NF            | VPACKSSDw, VPACKUSDw, VPSHUFd, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPERMD, VPERMPS, VPERMPD, VPERMQ,   |
|                      | VALIGND, VALIGNQ, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VSHUFI32X4, VSHUFI64X2, VSHUFF32X4, VSHUFF64X2, VPMULTISHIFTQB   |
| E4NF.nb <sup>2</sup> | VDBPSADBw, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDwD, VPMADDUBSw, VMOVSHDUP, VMOVSLDUP, VPSADBw, VPSHUFb, VPSHUFHW, VPSHUFw, VPSLLDQ, VPSRLDQ, VPSLLw, VPSRAW, VPSRLw, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ) <sup>3</sup> , VPUNPCKHBw, VPUNPCKHwD, VPUNPCKLBw, VPUNPCKLwD, VPERMw, VPERMI2w, VPERMT2w, VPERMB, VPERMI2B, VPERMT2B   |
| Type E5              | VCVTDQ2PD, PMOVsXBw, PMOVsXBW, PMOVsXBD, PMOVsXBQ, PMOVsXWD, PMOVsXWQ, PMOVsXDQ, PMOVzXBw, PMOVzXBD, PMOVzXBQ, PMOVzXWD, PMOVzXWQ, PMOVzXDQ  |
|                      | VCVTUDQ2PD   |
| Type E5NF            | VMOVDDUP   |

Table 4-14. EVEX Instructions in each Exception Class(Continued)

| Exception Class            | Instruction  |
|----------------------------|--|
| Type E6                    | VBROADCASTSS, VBROADCASTSD, VBROADCASTF32X4, VBROADCASTI32X4, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ,  |
|                            | VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VFPCLASSSD, VFPCLASSSS, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW |
| Type E6NF                  | VEEXTRACTF32X4, VEEXTRACTF64X2, VEEXTRACTF32X8, VINSERTF32X4, VINSERTF64X2, VINSERTF64X4, VINSERTF32X8, VINSERTI32X4, VINSERTI64X2, VINSERTI64X4, VINSERTI32X8, VEEXTRACTI32X4, VEEXTRACTI64X2, VEEXTRACTI32X8, VEEXTRACTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D, VPMOVWB, VPMOVSWB, VPMOVUSWB   |
| Type E7NM.128 <sup>4</sup> | VMOVLHPS, VMOVHLPS   |
| Type E7NM.                 | (VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW) <sup>5</sup> , VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVMB2M, VPMOVD2M, VPMOVQ2M, VPMOVW2M  |
| Type E9NF                  | VEEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ   |
| Type E10                   | VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS,  |
| Type E10NF                 | (VCVTSI2SD, VCVTUSI2SD) <sup>6</sup>   |
| Type E11                   | VCVTPH2PS, VCVTPS2PH   |
| Type E12                   | VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS   |
| Type E12NP                 | VGATHERPFODPD, VGATHERPFODPS, VGATHERPFOQPD, VGATHERPFOQPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPFODPD, VSCATTERPFODPS, VSCATTERPFOQPD, VSCATTERPFOQPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS   |

## NOTES:

1. Operand encoding FVI tupletype with immediate.
2. Embedded broadcast is not supported with the ".nb" suffix.
3. Operand encoding M128 tupletype.
4. #UD raised if EVEX.L'L !=00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.



### 4.10.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

**Table 4-15. Type E1 Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 4-8 not met,</li> <li>Opcode independent #UD condition in Table 4-9,</li> <li>Operand encoding #UD conditions in Table 4-10,</li> <li>Opmask encoding #UD condition of Table 4-11,</li> <li>If EVEX.b != 0,</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           | X      | EVEX.512: Memory operand is not 64-byte aligned.<br>EVEX.256: Memory operand is not 32-byte aligned.<br>EVEX.128: Memory operand is not 16-byte aligned.  |
|                            |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |

EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

**Table 4-16. Type E1NF Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | For an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0) |      |               | X                           | X      | EVEX.512: Memory operand is not 64-byte aligned.<br>EVEX.256: Memory operand is not 32-byte aligned.<br>EVEX.128: Memory operand is not 16-byte aligned.  |
|                            |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                            | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | For a page fault.   |

## 4.10.2 Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2.

**Table 4-17. Type E2 Class Exception Conditions**

| Exception                          | Real | Virtual 8086 | Protected and Compatibility | 64-bit | Cause of Exception   |
|------------------------------------|------|--------------|-----------------------------|--------|--|
| Invalid Opcode, #UD                | X    | X            |                             |        | If EVEX prefix present.  |
|                                    | X    | X            | X                           | X      | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.   |
|                                    |      |              | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                                    | X    | X            | X                           | X      | If preceded by a LOCK prefix (F0H).  |
|                                    |      |              | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.  |
|                                    | X    | X            | X                           | X      | If any corresponding CPUID feature flag is '0'.  |
| Device Not Available, #NM          | X    | X            | X                           | X      | If CR0.TS[bit 3]=1.  |
| Stack, SS(0)                       |      |              | X                           |        | If fault suppression not set, and an illegal address in the SS segment.  |
|                                    |      |              |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0)         |      |              | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  |
|                                    |      |              |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.   |
|                                    | X    | X            |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF(fault-code)         |      | X            | X                           | X      | If fault suppression not set, and a page fault.  |
| SIMD Floating-point Exception, #XM | X    | X            | X                           | X      | If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.   |

### 4.10.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

**Table 4-18. Type E3 Class Exception Conditions**

| Exception                          | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|------------------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD                | X    | X             |                             |        | If EVEX prefix present.   |
|                                    | X    | X             | X                           | X      | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.  |
|                                    |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0.</li> </ul> |
|                                    | X    | X             | X                           | X      | If preceded by a LOCK prefix (F0H).   |
|                                    |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                                    | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM          | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)                       |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                                    |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0)         |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                                    |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                                    | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code)         |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |
| Alignment Check #AC(0)             |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |
| SIMD Floating-point Exception, #XM | X    | X             | X                           | X      | If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.  |

EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

**Table 4-19. Type E3NF Class Exception Conditions**

| Exception                          | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|------------------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD                | X    | X             |                             |        | EVEX prefix.  |
|                                    | X    | X             | X                           | X      | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.  |
|                                    |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0.</li> </ul> |
|                                    | X    | X             | X                           | X      | If preceded by a LOCK prefix (F0H).   |
|                                    |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                                    | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM          | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)                       |      |               | X                           |        | For an illegal address in the SS segment.   |
|                                    |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0)         |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                                    |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                                    | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF(fault-code)         |      | X             | X                           | X      | For a page fault.   |
| Alignment Check #AC(0)             |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |
| SIMD Floating-point Exception, #XM | X    | X             | X                           | X      | If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.  |

### 4.10.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

**Table 4-20. Type E4 Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0 and in E4.nb subclass (see E4.nb entries in Table 4-14),</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |

EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

**Table 4-21. Type E4NF Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 4-8 not met,</li> <li>Opcode independent #UD condition in Table 4-9,</li> <li>Operand encoding #UD conditions in Table 4-10,</li> <li>Opmask encoding #UD condition of Table 4-11,</li> <li>If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 4-14),</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (F0H).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | For an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0) |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                            | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | For a page fault.   |

### 4.10.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

**Table 4-22. Type E5 Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |



EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

**Table 4-23. Type E5NF Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If an illegal address in the SS segment.  |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0) |      |               | X                           |        | If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                            | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | For a page fault.   |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |

### 4.10.6 Exceptions Type E6 and E6NF

Table 4-24. Type E6 Class Exception Conditions

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            |      |               | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            |      |               | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  |      |               | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
| Page Fault #PF(fault-code) |      |               | X                           | X      | If fault suppression not set, and a page fault.   |
| Alignment Check #AC(0)     |      |               | X                           | X      | For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |

EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

**Table 4-25. Type E6NF Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            |      |               | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            |      |               | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  |      |               | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | For an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0) |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.   |
| Page Fault #PF(fault-code) |      |               | X                           | X      | For a page fault.   |
| Alignment Check #AC(0)     |      |               | X                           | X      | For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |

### 4.10.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

**Table 4-26. Type E7NM Class Exception Conditions**

| Exception                 | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|---------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD       | X    | X             |                             |        | If EVEX prefix present.   |
|                           |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ Instruction specific EVEX.L'L restriction not met.</li> </ul> |
|                           | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                           |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                           | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM |      |               | X                           | X      | If CR0.TS[bit 3]=1.   |

## 4.10.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

**Table 4-27. Type E9 Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 4-8 not met,</li> <li>Opcode independent #UD condition in Table 4-9,</li> <li>Operand encoding #UD conditions in Table 4-10,</li> <li>Opmask encoding #UD condition of Table 4-11,</li> <li>If EVEX.b != 0,</li> <li>If EVEX.L'L != 00b (VL=128).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |

EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

**Table 4-28. Type E9NF Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 00b (VL=128).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If an illegal address in the SS segment.  |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0) |      |               | X                           |        | If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                            | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | For a page fault.   |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  |

### 4.10.9 Exceptions Type E10

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding and do not cause no SIMD FP exception, support memory fault suppression follow exception class E10.

**Table 4-29. Type E10 Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 4-8 not met,</li> <li>Opcode independent #UD condition in Table 4-9,</li> <li>Operand encoding #UD conditions in Table 4-10,</li> <li>Opmask encoding #UD condition of Table 4-11,</li> <li>If EVEX.b != 0.</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |

EVEX-encoded scalar instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E10NF.

**Table 4-30. Type E10NF Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0.</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.   |



### 4.10.10 Exception Type E11 (EVEX-only, mem arg no AC, floating-point exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11.

**Table 4-31. Type E11 Class Exception Conditions**

| Exception                          | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|------------------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD                | X    | X             |                             |        | If EVEX prefix present.   |
|                                    |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 4-8 not met,</li> <li>Opcode independent #UD condition in Table 4-9,</li> <li>Operand encoding #UD conditions in Table 4-10,</li> <li>Opmask encoding #UD condition of Table 4-11,</li> <li>If EVEX.b != 0,</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                                    | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                                    |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.   |
|                                    | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
| Device Not Available, #NM          | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)                       |      |               | X                           |        | If fault suppression not set, and an illegal address in the SS segment.   |
|                                    |      |               |                             | X      | If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0)         |      |               | X                           |        | If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                                    |      |               |                             | X      | If fault suppression not set, and the memory address is in a non-canonical form.  |
|                                    | X    | X             |                             |        | If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF (fault-code)        |      | X             | X                           | X      | If fault suppression not set, and a page fault.   |
| SIMD Floating-Point Exception, #XM | X    | X             | X                           | X      | If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.   |

### 4.10.11 Exception Type E12 and E12NP (VSIB mem arg, no AC, no floating-point exceptions)

Table 4-32. Type E12 Class Exception Conditions

| Exception                   | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|-----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD         | X    | X             |                             |        | If EVEX prefix present.   |
|                             |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                             | X    | X             | X                           | X      | If preceded by a LOCK prefix (F0H).   |
|                             |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a VEX prefix.  |
|                             | X    | X             | X                           | NA     | If address size attribute is 16 bit.  |
|                             | X    | X             | X                           | X      | If ModR/M.mod = '11b'.  |
|                             | X    | X             | X                           | X      | If ModR/M.rm != '100b'.   |
|                             | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
|                             | X    | X             | X                           | X      | If k0 is used (gather or scatter operation).  |
| Device Not Available, #NM   | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
|                             |      |               | X                           |        | For an illegal address in the SS segment.   |
| Stack, SS(0)                |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
|                             |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
| General Protection, #GP(0)  |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                             | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |
| Page Fault #PF (fault-code) |      | X             | X                           | X      | For a page fault.   |

EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

**Table 4-33. Type E12NP Class Exception Conditions**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception  |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD        | X    | X             |                             |        | If EVEX prefix present.   |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10,</li> <li>▪ Opmask encoding #UD condition of Table 4-11,</li> <li>▪ If EVEX.b != 0,</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul> |
|                            | X    | X             | X                           | X      | If preceded by a LOCK prefix (FOH).   |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a VEX prefix.  |
|                            | X    | X             | X                           | NA     | If address size attribute is 16 bit.  |
|                            | X    | X             | X                           | X      | If ModR/M.mod = '11b'.  |
|                            | X    | X             | X                           | X      | If ModR/M.rm != '100b'.   |
|                            | X    | X             | X                           | X      | If any corresponding CPUID feature flag is '0'.   |
|                            | X    | X             | X                           | X      | If k0 is used (gather or scatter operation).  |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.   |
| Stack, SS(0)               |      |               | X                           |        | For an illegal address in the SS segment.   |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.  |
| General Protection, #GP(0) |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.   |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.   |
|                            | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |

## 4.11 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

The exception behavior of VEX-encoded opmask instructions are listed below.

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

**Table 4-34. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)**

| Exception                 | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception   |
|---------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD       | X    | X             | X                           | X      | If relevant CPUID feature flag is '0'.   |
|                           | X    | X             |                             |        | If a VEX prefix is present.  |
|                           |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10.</li> </ul> |
|                           |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a VEX prefix.   |
|                           |      |               | X                           | X      | If ModRM:[7:6] != 11b.   |
| Device Not Available, #NM | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.  |

Exception conditions of Opmask instructions that address memory are listed as Type K21.

**Table 4-35. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)**

| Exception                  | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception   |
|----------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD        | X    | X             | X                           | X      | If relevant CPUID feature flag is '0'.   |
|                            | X    | X             |                             |        | If a VEX prefix is present.  |
|                            |      |               | X                           | X      | If CR4.OSXSAVE[bit 18]=0.<br>If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 4-8 not met,</li> <li>▪ Opcode independent #UD condition in Table 4-9,</li> <li>▪ Operand encoding #UD conditions in Table 4-10.</li> </ul> |
| Device Not Available, #NM  | X    | X             | X                           | X      | If CR0.TS[bit 3]=1.  |
|                            |      |               | X                           | X      | If any REX, F2, F3, or 66 prefixes precede a VEX prefix.   |
| Stack, SS(0)               | X    | X             | X                           |        | For an illegal address in the SS segment.  |
|                            |      |               |                             | X      | If a memory address referencing the SS segment is in a non-canonical form.   |
| General Protection, #GP(0) |      |               | X                           |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.   |
|                            |      |               |                             | X      | If the memory address is in a non-canonical form.  |
|                            | X    | X             |                             |        | If any part of the operand lies outside the effective address space from 0 to FFFFH.   |
| Page Fault #PF(fault-code) |      | X             | X                           | X      | For a page fault.  |
| Alignment Check #AC(0)     |      | X             | X                           | X      | If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.  |

This page was intentionally left blank.

## CHAPTER 5

# INSTRUCTION SET REFERENCE, A-Z

---

Instructions described in this document follow the general documentation convention established in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* and *2B*. Additional notations and conventions adopted in this document are listed in *Section 5.1*. *Section 5.1.5.1* covers supplemental information that applies to a specific subset of instructions.

## 5.1 INTERPRETING INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections that are outside of those conventions described in *Section 3.1* of the *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*.

### 5.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The table below provides an example summary table:

**ADDPS—Add Packed Single-Precision Floating-Point Values (THIS IS AN EXAMPLE)**

| Opcode/<br>Instruction   | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------------------------|--------------------------|--|
| OF 58 /r<br>ADDPS xmm1, xmm2/m128  | V/V                          | SSE                      | Add packed single-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.                              |
| VEX.NDS.128.OF 58 /r<br>VADDPS xmm1,xmm2, xmm3/m128                              | V/V                          | AVX                      | Add packed single-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.                              |
| VEX.NDS.256.OF 58 /r<br>VADDPS ymm1, ymm2, ymm3/m256                             | V/V                          | AVX                      | Add packed single-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.                              |
| VEX.L1.OF.W0 41 /r<br>KANDW k1, k2, k3   | V/V                          | AVX512F                  | Bitwise AND word masks k2 and k3 and place result in k1.   |
| EVEX.NDS.128.OF.W0 58 /r<br>VADDPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | V/V                          | AVX512VL<br>AVX512F      | Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.   |
| EVEX.NDS.256.OF.W0 58 /r<br>VADDPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | V/V                          | AVX512VL<br>AVX512F      | Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.   |
| EVEX.NDS.512.OF.W0 58 /r<br>VADDPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er} | V/V                          | AVX512F                  | Add packed single-precision floating-point values from zmm3/m512/m32bcst with zmm2 and store result in zmm1 with writemask k1. |

**5.1.2 Opcode Column in the Instruction Summary Table**

For notation and conventions applicable to instructions that do not use VEX or EVEX prefixes, consult *Section 3.1* of the *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*.

In the Instruction Summary Table, the Opcode column presents each instruction encoded using the VEX prefix in following form (including the modR/M byte if applicable, the immediate byte if applicable):

**VEX.[NDS/NDD/DS].[128,256,LO,L1,LIG].[66,F2,F3].OF/OF3A/OF38.[W0,W1,WIG] opcode [/r]  
[ib,/is4]**

- **VEX:** indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B.

The encoding of various sub-fields of the VEX prefix is described using the following notations:

- **NDS, NDD, DDS:** implies that VEX.vvvv field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result. If NDS, NDD and DDS are absent (i.e. VEX.vvvv does not encode an operand), VEX.vvvv must be 1111b.
- **128,256,LO,L1:** VEX.L fields can be 0 (denoted by VEX.128 or VEX.L0 for mask instructions) or 1 (denoted by VEX.256 or VEX.L1 for mask instructions). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:



- If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L = 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
  - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Three situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L = 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L = 1 by causing an #UD exception (e.g. VMOVLPS); (c) For VEX-encoded, scalar, SIMD floating-point instructions, software should encode the instruction with VEX.L = 0 to ensure software compatibility with future processor generations. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions, except VBROADCASTSx are unique cases.
  - VEX.L0 and VEX.L1 notations are used in the case of masking instructions such as KANDW since the VEX.L bit is not used to distinguish between the 128-bit and 256-bit forms for these instructions. Instead, this bit is used to distinguish between the two operand form (VEX.L0) and the three operand form (VEX.L1) of the same mask instruction.
  - If VEX.L0 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 0. An attempt to encode this instruction with VEX.L = 1 can result in one of two situations: (a) if VEX.L1 version is defined, the processor will behave according to the defined VEX.L1 behavior; (b) an #UD occurs if there is no VEX.L1 version defined.
  - If VEX.L1 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L = 0 can result in one of two situations: (a) if VEX.L0 version is defined, the processor will behave according to the defined VEX.L1 behavior; (b) an #UD occurs if there is no VEX.L0 version defined.
  - **LIG:** VEX.L bit ignored
- **66,F2,F3:** The presence or absence of these value maps to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.
  - **0F,0F3A,0F38:** The presence maps to a valid encoding of the VEX.mmmmm field. Only three encoded values of VEX.mmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid VEX.mmmmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmm field must be encoded using the 3-byte form of VEX prefix.
  - **0F,0F3A,0F38 and 2-byte/3-byte VEX.** The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.
  - **W0:** VEX.W=0.
  - **W1:** VEX.W=1.
  - **WIG:** VEX.W bit ignored
  - The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. If neither W0 or W1 is present, the instruction may be encoded using either the two-byte form (if the opcode semantic does

not require VEX subfields not present in the two-byte form of VEX) or the three-byte form of VEX. Encoding an instruction using the two-byte form of VEX is equivalent to W0.

- **opcode:** Instruction opcode.
  - **ib:** An 8-bit immediate byte is present and used as one of the instructions operands.
  - **/is4:** An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].
  - **imz2:** Part of the is4 immediate byte provides control functions that apply to two-source permute instructions
- In general, the encoding of VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column.

#### **EVEX.[NDS/NDD/DDS].[128,256,512,LIG].[66,F2,F3].OF/OF3A/OF38.[W0,W1,WIG] opcode [/r] [ib,/is4]**

- **EVEX:** The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 4.2 for more detail on the EVEX prefix.

The encoding of various sub-fields of the EVEX prefix is described using the following notations:

- **NDS, NDD, DDS:** implies that EVEX.vvvv (and EVEX.v') field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result. If both NDS and NDD absent (i.e. EVEX.vvvv does not encode an operand), EVEX.vvvv must be 1111b (and EVEX.v' must be 1b).
- **128, 256, 512, LIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.
- **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.
- **OF,OF3A,OF38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.
- **W0:** EVEX.W=0.
- **W1:** EVEX.W=1.
- **WIG:** EVEX.W bit ignored
- **opcode:** Instruction opcode.
- **/is4:** An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].
- **imz2:** Part of the is4 immediate byte provides control functions that apply to two-source permute instructions
- In general, the encoding of EVEX.R and R', EVEX.X and X', and EVEX.B and B' fields are not shown explicitly in the opcode column.

### 5.1.3 Instruction Column in the Instruction Summary Table

- **xmm** — an XMM register. The XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available in 64-bit mode. XMM16 through XMM31 are available in 64-bit mode via EVEX prefix.
- **ymm** — a YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode. YMM16 through YMM31 are available in 64-bit mode via EVEX prefix.
- **m256** — A 32-byte operand in memory.
- **ymm/m256** - a YMM register or 256-bit memory operand.

- **<YMMO>**: indicates use of the YMM0 register as an implicit argument.
- **zmm** — a ZMM register. The 512-bit ZMM registers require EVEX prefix and are: ZMM0 through ZMM7; ZMM8 through ZMM31 are available in 64-bit mode.
- **m512** — A 64-byte operand in memory.
- **zmm/m512** — a ZMM register or 512-bit memory operand.
- **{k1}{z}** — a mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — a mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.
- **mV** — a vector memory operand; the operand size is dependent on the instruction.
- **vm32{x,y,z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- **vm64{x,y,z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).
- **zmm/m512/m32bcst** — an operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — an operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
- **<ZMMO>**: indicates use of the ZMM0 register as an implicit argument.
- **{er}** indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** - Denotes the first source operand in the instruction syntax of an instruction encoded with the EVEX prefix and having two or more source operands.
- **SRC2** - Denotes the second source operand in the instruction syntax of an instruction encoded with the EVEX prefix and having two or more source operands.
- **SRC3** - Denotes the third source operand in the instruction syntax of an instruction encoded with the EVEX prefix and having three source operands.
- **SRC** - The source in a single-source instruction.
- **DST** - the destination in an instruction. This field is encoded by reg\_field.

### 5.1.4 64/32 bit Mode Support column in the Instruction Summary Table

The “64/32 bit Mode Support” column in the Instruction Summary table indicates whether an opcode sequence is supported in 64-bit or the Compatibility/other IA32 modes.

The 64-bit mode support is to the left of the ‘slash’ and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E.** — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence

of valid instructions in other modes).

- **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
- **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
- **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.

The compatibility/Legacy mode support is to the right of the 'slash' and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E.** — Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

### 5.1.5 CPUID Support column in the Instruction Summary Table

The fourth column holds abbreviated CPUID feature flags (e.g. appropriate bits in CPUID.1:ECX, CPUID.1:EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AVX/F16C support; bits in CPUID.(EAX=07H,ECX=0):BCX for AVX2/AVX512F etc) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

For entries that reference to CPUID feature flags listed in Table 2-1, software should follow the detection procedure described in Section 2.1 and Section 2.2.

For entries that reference to CPUID feature flags listed in Table 2-1 and AVX512VL, software should follow the detection procedure described in Section 2.3.

#### 5.1.5.1 Operand Encoding Column in the Instruction Summary Table

The "operand encoding" column is abbreviated as Op/En in the Instruction Summary table heading. Instruction operand encoding information is provided for each assembly instruction syntax using a letter to cross reference to a row entry in the operand encoding definition table that follows the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

EVEX encoded instructions employ compressed  $\text{disp8} * N$  encoding of the displacement bytes, where N is defined in Table 4-5 and Table 4-6, according to tuple types. The Op/En column of an EVEX encoded instruction uses an abbreviation that corresponds to the tuple type abbreviation (and may include an additional abbreviation related to ModR/M and vvvv encoding). Most EVEX encoded instructions with VEX encoded equivalent have the ModR/M and vvvv encoding order. In such cases, the Tuple abbreviation is shown and the ModR/M, vvvv encoding abbreviation may be omitted.

#### NOTES

The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.

In the encoding definition table, the letter 'r' within a pair of parentheses denotes the content of the operand will be read by the processor. The letter 'w' within a pair of parenthesis denotes the content of the operand will be updated by the processor.

## 5.2 SUMMARY OF TERMS

- **"Legacy SSE"**: Refers to SSE, SSE2, SSE3, SSSE3, SSE4, and any future instruction sets referencing XMM registers and encoded without a VEX or EVEX prefix.

- **XGETBV, XSETBV, XSAVE, XRSTOR** are defined in *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **VEX**: refers to a two-byte or three-byte prefix. AVX and FMA instructions are encoded using a VEX prefix.
- **EVEX**: refers to a four-byte prefix. AVX512F instructions are encoded using an EVEX prefix.
- **VEX.vvvv**. The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm\_field**: shorthand for the ModR/M *r/m* field and any REX.B
- **reg\_field**: shorthand for the ModR/M *reg* field and any REX.R

## 5.3 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a shorthand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one or more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator 'not' is expressed using '!'. Additionally, the conditional expression "A?B:C" expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g. andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g. andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g. norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g. zeros or norCB.

The 3-input expression "majorABC" returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression "minorABC" returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include;

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.

Table 5-1. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

| Imm   | [7:4]      |                |                |                |                |                |                 |                 |
|-------|------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|
| [3:0] | 0H         | 1H             | 2H             | 3H             | 4H             | 5H             | 6H              | 7H              |
| 00H   | FALSE      | andAnorBC      | norBnandAC     | andA!B         | norCnandBA     | andA!C         | andAxorBC       | andAnandBC      |
| 01H   | norABC     | norCB          | norBxorAC      | A?!B:norBC     | norCxorBA      | A?!C:norBC     | A?xorBC:norBC   | A?nandBC:norBC  |
| 02H   | andCnorBA  | norBxnorAC     | andC!B         | norBnorAC      | C?norBA:andBA  | C?norBA:A      | C?!B:andBA      | C?!B:A          |
| 03H   | norBA      | norBandAC      | C?!B:norBA     | !B             | C?norBA:xnorBA | A?!C:!B        | A?xorBC:!B      | A?nandBC:!B     |
| 04H   | andBnorAC  | norCxnorBA     | B?norAC:andAC  | B?norAC:A      | andB!C         | norCnorBA      | B?!C:andAC      | B?!C:A          |
| 05H   | norCA      | norCandBA      | B?norAC:xnorAC | A?!B:!C        | B?!C:norAC     | !C             | A?xorBC:!C      | A?nandBC:!C     |
| 06H   | norAxnorBC | A?norBC:xorBC  | B?norAC:C      | xorBorAC       | C?norBA:B      | xorCorBA       | xorCB           | B?!C:orAC       |
| 07H   | norAandBC  | minorABC       | C?!B:!A        | nandBorAC      | B?!C:!A        | nandCorBA      | A?xorBC:nandBC  | nandCB          |
| 08H   | norAnandBC | A?norBC:andBC  | andCxorBA      | A?!B:andBC     | andBxorAC      | A?!C:andBC     | A?xorBC:andBC   | xorAandBC       |
| 09H   | norAxorBC  | A?norBC:xnorBC | C?xorBA:norBA  | A?!B:xnorBC    | B?xorAC:norAC  | A?!C:xnorBC    | xnorABC         | A?nandBC:xnorBC |
| 0AH   | andCIA     | A?norBC:C      | andCnandBA     | A?!B:C         | C?!A:andBA     | xorCA          | xorCandBA       | A?nandBC:C      |
| 0BH   | C?!A:norBA | C?!A:!B        | C?nandBA:norBA | C?nandBA:!B    | B?xorAC:!A     | B?xorAC:nandAC | C?nandBA:xnorBA | nandBxnorAC     |
| 0CH   | andB!A     | A?norBC:B      | B?!A:andAC     | xorBA          | andBnandAC     | A?!C:B         | xorBandAC       | A?nandBC:B      |
| 0DH   | B?!A:norAC | B?!A:!C        | B?!A:xnorAC    | C?xorBA:nandBA | B?nandAC:norAC | B?nandAC:!C    | B?nandAC:xnorAC | nandCxnorBA     |
| 0EH   | norAnorBC  | xorAorBC       | B?!A:C         | A?!B:orBC      | C?!A:B         | A?!C:orBC      | B?nandAC:C      | A?nandBC:orBC   |
| 0FH   | !A         | nandAorBC      | C?nandBA:!A    | nandBA         | B?nandAC:!A    | nandCA         | nandAxnorBC     | nandABC         |

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.

Table 5-2. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

| Imm   | [7:4]                |                        |                       |                      |                       |                      |                      |                   |
|-------|----------------------|------------------------|-----------------------|----------------------|-----------------------|----------------------|----------------------|-------------------|
| [3:0] | 08H                  | 09H                    | 0AH                   | 0BH                  | 0CH                   | 0DH                  | 0EH                  | 0FH               |
| 00H   | <i>andABC</i>        | <i>andAxnorBC</i>      | <i>andCA</i>          | <i>B?andAC:A</i>     | <i>andBA</i>          | <i>C?andBA:A</i>     | <i>andAorBC</i>      | <i>A</i>          |
| 01H   | <i>A?andBC:norBC</i> | <i>B?andAC:!C</i>      | <i>A?C:norBC</i>      | <i>C?A:!B</i>        | <i>A?B:norBC</i>      | <i>B?A:!C</i>        | <i>xnorAorBC</i>     | <i>orAnorBC</i>   |
| 02H   | <i>andCxnorBA</i>    | <i>B?andAC:xorAC</i>   | <i>B?andAC:C</i>      | <i>B?andAC:orAC</i>  | <i>C?xnorBA:andBA</i> | <i>B?A:xorAC</i>     | <i>B?A:C</i>         | <i>B?A:orAC</i>   |
| 03H   | <i>A?andBC:!B</i>    | <i>xnorBandAC</i>      | <i>A?C:!B</i>         | <i>nandBnandAC</i>   | <i>xnorBA</i>         | <i>B?A:nandAC</i>    | <i>A?orBC:!B</i>     | <i>orA!B</i>      |
| 04H   | <i>andBxnorAC</i>    | <i>C?andBA:xorBA</i>   | <i>B?xnorAC:andAC</i> | <i>B?xnorAC:A</i>    | <i>C?andBA:B</i>      | <i>C?andBA:orBA</i>  | <i>C?A:B</i>         | <i>C?A:orBA</i>   |
| 05H   | <i>A?andBC:!C</i>    | <i>xnorCandBA</i>      | <i>xnorCA</i>         | <i>C?A:nandBA</i>    | <i>A?B:!C</i>         | <i>nandCnandBA</i>   | <i>A?orBC:!C</i>     | <i>orA!C</i>      |
| 06H   | <i>A?andBC:xorBC</i> | <i>xorABC</i>          | <i>A?C:xorBC</i>      | <i>B?xnorAC:orAC</i> | <i>A?B:xorBC</i>      | <i>C?xnorBA:orBA</i> | <i>A?orBC:xorBC</i>  | <i>orAxorBC</i>   |
| 07H   | <i>xnorAandBC</i>    | <i>A?xnorBC:nandBC</i> | <i>A?C:nandBC</i>     | <i>nandBxorAC</i>    | <i>A?B:nandBC</i>     | <i>nandCxorBA</i>    | <i>A?orBCnandBC</i>  | <i>orAnandBC</i>  |
| 08H   | <i>andCB</i>         | <i>A?xnorBC:andBC</i>  | <i>andCorAB</i>       | <i>B?C:A</i>         | <i>andBorAC</i>       | <i>C?B:A</i>         | <i>majorABC</i>      | <i>orAandBC</i>   |
| 09H   | <i>B?C:norAC</i>     | <i>xnorCB</i>          | <i>xnorCorBA</i>      | <i>C?orBA:!B</i>     | <i>xnorBorAC</i>      | <i>B?orAC:!C</i>     | <i>A?orBC:xnorBC</i> | <i>orAxnorBC</i>  |
| 0AH   | <i>A?andBC:C</i>     | <i>A?xnorBC:C</i>      | <i>C</i>              | <i>B?C:orAC</i>      | <i>A?B:C</i>          | <i>B?orAC:xorAC</i>  | <i>orCandBA</i>      | <i>orCA</i>       |
| 0BH   | <i>B?C:!A</i>        | <i>B?C:nandAC</i>      | <i>orCnorBA</i>       | <i>orC!B</i>         | <i>B?orAC:!A</i>      | <i>B?orAC:nandAC</i> | <i>orCxnorBA</i>     | <i>nandBnorAC</i> |
| 0CH   | <i>A?andBC:B</i>     | <i>A?xnorBC:B</i>      | <i>A?C:B</i>          | <i>C?orBA:xorBA</i>  | <i>B</i>              | <i>C?B:orBA</i>      | <i>orBandAC</i>      | <i>orBA</i>       |
| 0DH   | <i>C?B!A</i>         | <i>C?B:nandBA</i>      | <i>C?orBA:!A</i>      | <i>C?orBA:nandBA</i> | <i>orBnorAC</i>       | <i>orB!C</i>         | <i>orBxnorAC</i>     | <i>nandCnorBA</i> |
| 0EH   | <i>A?andBC:orBC</i>  | <i>A?xnorBC:orBC</i>   | <i>A?C:orBC</i>       | <i>orCxorBA</i>      | <i>A?B:orBC</i>       | <i>orBxorAC</i>      | <i>orCB</i>          | <i>orABC</i>      |
| 0FH   | <i>nandAnandBC</i>   | <i>nandAxorBC</i>      | <i>orCIA</i>          | <i>orCnandBA</i>     | <i>orB!A</i>          | <i>orBnandAC</i>     | <i>nandAnorBC</i>    | <i>TRUE</i>       |

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result\_imm8 = Table\_Lookup\_Entry( 0F0H, 0CCH, 0AAH)

Table\_Lookup\_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.

## 5.4 INSTRUCTION SET REFERENCE

<Only instructions modified by AVX512F are included.>

**ADDPD—Add Packed Double-Precision Floating-Point Values**

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F 58 /r<br>ADDPD xmm1, xmm2/m128   | RM         | V/V                          | SSE2                     | Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.                            |
| VEX.NDS.128.66.0F.WIG 58 /r<br>VADDPD xmm1,xmm2,<br>xmm3/m128                      | RVM        | V/V                          | AVX                      | Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.                            |
| VEX.NDS.256.66.0F.WIG 58 /r<br>VADDPD ymm1, ymm2,<br>ymm3/m256                     | RVM        | V/V                          | AVX                      | Add packed double-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.                            |
| EVEX.NDS.128.66.0F.W1 58 /r<br>VADDPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Add packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1. |
| EVEX.NDS.256.66.0F.W1 58 /r<br>VADDPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Add packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1. |
| EVEX.NDS.512.66.0F.W1 58 /r<br>VADDPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Add packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1. |

**Instruction Operand Encoding**

| Op/En  | 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      | ModRM:r/m (r) | NA        |
| FV-RVM | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

**Description**

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

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

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

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

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



**Operation****VADDPD (EVEX encoded versions) when src2 operand is a vector register**

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

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

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VADDPD (EVEX encoded versions) when src2 operand is a memory source**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VADDPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

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

DEST[191:128] ← SRC1[191:128] + SRC2[191:128]

DEST[255:192] ← SRC1[255:192] + SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VADDPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

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

DEST[MAX\_VL-1:128] ← 0

**ADDPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] + SRC[63:0]

DEST[127:64] ← DEST[127:64] + SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDPD \_\_m512d \_\_mm512\_add\_pd (\_\_m512d a, \_\_m512d b);

VADDPD \_\_m512d \_\_mm512\_mask\_add\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VADDPD \_\_m512d \_\_mm512\_maskz\_add\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VADDPD \_\_m256d \_\_mm256\_mask\_add\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VADDPD \_\_m256d \_\_mm256\_maskz\_add\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VADDPD \_\_m128d \_\_mm\_mask\_add\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VADDPD \_\_m128d \_\_mm\_maskz\_add\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VADDPD \_\_m512d \_\_mm512\_add\_round\_pd (\_\_m512d a, \_\_m512d b, int);

VADDPD \_\_m512d \_\_mm512\_mask\_add\_round\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VADDPD \_\_m512d \_\_mm512\_maskz\_add\_round\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

ADDPD \_\_m256d \_\_mm256\_add\_pd (\_\_m256d a, \_\_m256d b);

ADDPD \_\_m128d \_\_mm\_add\_pd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## ADDPS—Add Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| OF 58 /r<br>ADDPS xmm1, xmm2/m128  | RM         | V/V                          | SSE                      | Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.                           |
| VEX.NDS.128.OF.WIG 58 /r<br>VADDPS xmm1,xmm2,<br>xmm3/m128                       | RVM        | V/V                          | AVX                      | Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.                           |
| VEX.NDS.256.OF.WIG 58 /r<br>VADDPS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | AVX                      | Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.                           |
| EVEX.NDS.128.OF.W0 58 /r<br>VADDPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | FV         | V/V                          | AVX512VL<br>AVX512F      | Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1. |
| EVEX.NDS.256.OF.W0 58 /r<br>VADDPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | FV         | V/V                          | AVX512VL<br>AVX512F      | Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1. |
| EVEX.NDS.512.OF.W0 58 /r<br>VADDPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er} | FV         | V/V                          | AVX512F                  | Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En  | 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      | ModRM:r/m (r) | NA        |
| FV-RVM | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

**Operation****VADDPS (EVEX encoded versions) when src2 operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

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

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VADDPS (EVEX encoded versions) when src2 operand is a memory source**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

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

ELSE

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

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VADDPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]

DEST[63:32] ← SRC1[63:32] + SRC2[63:32]

DEST[95:64] ← SRC1[95:64] + SRC2[95:64]

DEST[127:96] ← SRC1[127:96] + SRC2[127:96]

DEST[159:128] ← SRC1[159:128] + SRC2[159:128]

DEST[191:160] ← SRC1[191:160] + SRC2[191:160]

DEST[223:192] ← SRC1[223:192] + SRC2[223:192]

DEST[255:224] ← SRC1[255:224] + SRC2[255:224].  
 DEST[MAX\_VL-1:256] ← 0

#### VADDPS (VEX.128 encoded version)

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

#### ADDPS (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[MAX\_VL-1:128] (Unmodified)

#### Intel C/C++ Compiler Intrinsic Equivalent

VADDPS \_\_m512 \_\_mm512\_add\_ps (\_\_m512 a, \_\_m512 b);  
 VADDPS \_\_m512 \_\_mm512\_mask\_add\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VADDPS \_\_m512 \_\_mm512\_maskz\_add\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VADDPS \_\_m256 \_\_mm256\_mask\_add\_ps (\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VADDPS \_\_m256 \_\_mm256\_maskz\_add\_ps (\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VADDPS \_\_m128 \_\_mm\_mask\_add\_ps (\_\_m128d s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VADDPS \_\_m128 \_\_mm\_maskz\_add\_ps (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VADDPS \_\_m512 \_\_mm512\_add\_round\_ps (\_\_m512 a, \_\_m512 b, int);  
 VADDPS \_\_m512 \_\_mm512\_mask\_add\_round\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VADDPS \_\_m512 \_\_mm512\_maskz\_add\_round\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 ADDPS \_\_m256 \_\_mm256\_add\_ps (\_\_m256 a, \_\_m256 b);  
 ADDPS \_\_m128 \_\_mm\_add\_ps (\_\_m128 a, \_\_m128 b);

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

#### Other Exceptions

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## ADDSD—Add Scalar Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| F2 0F 58 /r<br>ADDSD xmm1, xmm2/m64                                       | RM         | V/V                          | SSE2                     | Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.                   |
| VEX.NDS.128.F2.0F.WIG 58 /r<br>VADDSD xmm1, xmm2,<br>xmm3/m64             | RVM        | V/V                          | AVX                      | Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.                   |
| EVEX.NDS.LIG.F2.0F.W1 58 /r<br>VADDSD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En   | 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      | ModRM:r/m (r) | NA        |
| T1S-RVM | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Adds the low double-precision floating-point values from the second source operand and the first source operand and stores the double-precision floating-point result in the destination operand.

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

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

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

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

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

### Operation

#### VADDSD (EVEX encoded version)

IF (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

#### **VADDSD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

#### **ADDSD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] + SRC[63:0]

DEST[MAX\_VL-1:64] (Unmodified)

#### **Intel C/C++ Compiler Intrinsic Equivalent**

VADDSD \_\_m128d \_\_mm\_mask\_add\_sd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VADDSD \_\_m128d \_\_mm\_maskz\_add\_sd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VADDSD \_\_m128d \_\_mm\_add\_round\_sd (\_\_m128d a, \_\_m128d b, int);

VADDSD \_\_m128d \_\_mm\_mask\_add\_round\_sd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b, int);

VADDSD \_\_m128d \_\_mm\_maskz\_add\_round\_sd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b, int);

ADDSD \_\_m128d \_\_mm\_add\_sd (\_\_m128d a, \_\_m128d b);

#### **SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

#### **Other Exceptions**

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## ADDSS—Add Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F3 0F 58 /r<br>ADDSS xmm1, xmm2/m32                                      | RM         | V/V                          | SSE                      | Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.                   |
| VEX.NDS.128.F3.0F.WIG 58 /r<br>VADDSS xmm1, xmm2,<br>xmm3/m32            | RVM        | V/V                          | AVX                      | Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.                   |
| EVEX.NDS.LIG.F3.0F.W0 58 /r<br>VADDSS xmm1{k1}{z}, xmm2,<br>xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Adds the low single-precision floating-point values from the second source operand and the first source operand, and stores the double-precision floating-point result in the destination operand.

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

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

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

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

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

### Operation

#### VADDSS (EVEX encoded versions)

IF (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SRC1[31:0] + SRC2[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0



FI;

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

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

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

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

DEST[31:0] ← DEST[31:0] + SRC[31:0]

DEST[MAX\_VL-1:32] (Unmodified)

#### **Intel C/C++ Compiler Intrinsic Equivalent**

VADDSS \_\_m128 \_mm\_mask\_add\_ss (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VADDSS \_\_m128 \_mm\_maskz\_add\_ss (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VADDSS \_\_m128 \_mm\_add\_round\_ss (\_\_m128 a, \_\_m128 b, int);

VADDSS \_\_m128 \_mm\_mask\_add\_round\_ss (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int);

VADDSS \_\_m128 \_mm\_maskz\_add\_round\_ss (\_\_mmask8 k, \_\_m128 a, \_\_m128 b, int);

ADDSS \_\_m128 \_mm\_add\_ss (\_\_m128 a, \_\_m128 b);

#### **SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

#### **Other Exceptions**

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F3A.W0 03 /r ib<br>VALIGND xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift right and merge vectors xmm2 and<br>xmm3/m128/m32bcst with double-word granularity<br>using imm8 as number of elements to shift, and store the<br>final result in xmm1, under writemask. |
| EVEX.NDS.128.66.0F3A.W1 03 /r ib<br>VALIGNQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift right and merge vectors xmm2 and<br>xmm3/m128/m64bcst with quad-word granularity using<br>imm8 as number of elements to shift, and store the final<br>result in xmm1, under writemask.   |
| EVEX.NDS.256.66.0F3A.W0 03 /r ib<br>VALIGND ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift right and merge vectors ymm2 and<br>ymm3/m256/m32bcst with double-word granularity<br>using imm8 as number of elements to shift, and store the<br>final result in ymm1, under writemask. |
| EVEX.NDS.256.66.0F3A.W1 03 /r ib<br>VALIGNQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift right and merge vectors ymm2 and<br>ymm3/m256/m64bcst with quad-word granularity using<br>imm8 as number of elements to shift, and store the final<br>result in ymm1, under writemask.   |
| EVEX.NDS.512.66.0F3A.W0 03 /r ib<br>VALIGND zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | FV        | V/V                          | AVX512F                  | Shift right and merge vectors zmm2 and<br>zmm3/m512/m32bcst with double-word granularity<br>using imm8 as number of elements to shift, and store the<br>final result in zmm1, under writemask. |
| EVEX.NDS.512.66.0F3A.W1 03 /r ib<br>VALIGNQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | FV        | V/V                          | AVX512F                  | Shift right and merge vectors zmm2 and<br>zmm3/m512/m64bcst with quad-word granularity using<br>imm8 as number of elements to shift, and store the final<br>result in zmm1, under writemask.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).

### Operation

#### VALIGND (EVEX encoded versions)

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

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

THEN

FOR j ← 0 TO KL-1

i ← j \* 32

```

        src[j+31:i] ← SRC2[31:0]
    ENDFOR;
    ELSE src ← SRC2
FI
; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[2:0]
            ELSE SHIFT = imm8[3:0]
        FI
    FI;
tmp[2VL-1:0] ← tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← tmp[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VALIGNQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)  
 IF (SRC2 \*is memory\*) (AND EVEX.b = 1)

```

    THEN
        FOR j ← 0 TO KL-1
            i ← j * 64
            src[j+63:i] ← SRC2[63:0]
        ENDFOR;
    ELSE src ← SRC2
FI

```

```

; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right quadword elements
IF VL = 128
    THEN SHIFT = imm8[0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[1:0]
            ELSE SHIFT = imm8[2:0]
        FI
    FI;
FI;

```

```

tmp[2VL-1:0] ← tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← tmp[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VALIGND __m512i_mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i_mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i_mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i_mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i_mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i_mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i_mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i_mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i_mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i_mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i_mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i_mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i_mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i_mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);

```

### Exceptions

See Exceptions Type E4NF.

## VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F38.W1 65 /r<br>VBLENDMPD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.        |
| EVEX.NDS.256.66.0F38.W1 65 /r<br>VBLENDMPD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.        |
| EVEX.NDS.512.66.0F38.W1 65 /r<br>VBLENDMPD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.        |
| EVEX.NDS.128.66.0F38.W0 65 /r<br>VBLENDMPS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.        |
| EVEX.NDS.256.66.0F38.W0 65 /r<br>VBLENDMPS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.        |
| EVEX.NDS.512.66.0F38.W0 65 /r<br>VBLENDMPS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

### Operation

#### VBLENDMPD (EVEX encoded versions)

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no controlmask\*

    THEN

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

        THEN

```

        DEST[i+63:i] ← SRC2[63:0]
    ELSE
        DEST[i+63:i] ← SRC2[i+63:i]
    FI;
ELSE
    IF *merging-masking*           ; merging-masking
        THEN DEST[i+63:i] ← SRC1[i+63:i]
    ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
    FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VBLENDMPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN DEST[i+31:i] ← SRC1[i+31:i]
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VBLENDMPD __m512d __mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d __mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d __mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 __mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 __mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 __mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type E4.

## VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F38.W0 66 /r<br>VPBLENDMB xmm1 {k1}{z},<br>xmm2, xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask. |
| EVEX.NDS.256.66.0F38.W0 66 /r<br>VPBLENDMB ymm1 {k1}{z},<br>ymm2, ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask. |
| EVEX.NDS.512.66.0F38.W0 66 /r<br>VPBLENDMB zmm1 {k1}{z},<br>zmm2, zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask. |
| EVEX.NDS.128.66.0F38.W1 66 /r<br>VPBLENDMW xmm1 {k1}{z},<br>xmm2, xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask. |
| EVEX.NDS.256.66.0F38.W1 66 /r<br>VPBLENDMW ymm1 {k1}{z},<br>ymm2, ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask. |
| EVEX.NDS.512.66.0F38.W1 66 /r<br>VPBLENDMW zmm1 {k1}{z},<br>zmm2, zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).

### Operation

#### VPBLENDMB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 8$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+7:i] \leftarrow SRC2[i+7:i]$

ELSE

IF \*merging-masking\* ;merging-masking

THEN  $DEST[i+7:i] \leftarrow SRC1[i+7:i]$

```

        ELSE                                ; zeroing-masking
            DEST[i+7:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

**VPBLENDMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC2[i+15:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN DEST[i+15:i] ← SRC1[i+15:i]
        ELSE                                  ; zeroing-masking
            DEST[i+15:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPBLENDMB __m512i __mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i __mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i __mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i __mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i __mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i __mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.



## VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F38.W0 64 /r<br>VPBLENDMD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask. |
| EVEX.NDS.256.66.0F38.W0 64 /r<br>VPBLENDMD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask. |
| EVEX.NDS.512.66.0F38.W0 64 /r<br>VPBLENDMD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask. |
| EVEX.NDS.128.66.0F38.W1 64 /r<br>VPBLENDMQ xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.     |
| EVEX.NDS.256.66.0F38.W1 64 /r<br>VPBLENDMQ ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.     |
| EVEX.NDS.512.66.0F38.W1 64 /r<br>VPBLENDMQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.     |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

### Operation

#### VPBLENDMD (EVEX encoded versions)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no controlmask\*

  THEN

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

    THEN

```

        DEST[i+31:i] ← SRC2[31:0]
    ELSE
        DEST[i+31:i] ← SRC2[i+31:i]
    FI;
ELSE
    IF *merging-masking*           ; merging-masking
        THEN DEST[i+31:i] ← SRC1[i+31:i]
    ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
    FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

**VPBLENDMD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

```

    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN DEST[i+31:i] ← SRC1[i+31:i]
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI;
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBLENDMD \_\_m512i \_\_mm512\_mask\_blend\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPBLENDMD \_\_m256i \_\_mm256\_mask\_blend\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPBLENDMD \_\_m128i \_\_mm\_mask\_blend\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

VPBLENDMQ \_\_m512i \_\_mm512\_mask\_blend\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPBLENDMQ \_\_m256i \_\_mm256\_mask\_blend\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPBLENDMQ \_\_m128i \_\_mm\_mask\_blend\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

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

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

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

**Operation****VANDPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

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

THEN

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

ELSE

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

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE

; zeroing-masking

DEST[i+63:i] = 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VANDPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]

DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]

DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VANDPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

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

DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]

DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VANDPD \_\_m512d \_\_mm512\_and\_pd (\_\_m512d a, \_\_m512d b);

VANDPD \_\_m512d \_\_mm512\_mask\_and\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDPD \_\_m512d \_\_mm512\_maskz\_and\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDPD \_\_m256d \_\_mm256\_mask\_and\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDPD \_\_m256d \_\_mm256\_maskz\_and\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDPD \_\_m128d \_\_mm\_mask\_and\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDPD \_\_m128d \_\_mm\_maskz\_and\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDPD \_\_m256d \_\_mm256\_and\_pd (\_\_m256d a, \_\_m256d b);

ANDPD \_\_m128d \_\_mm\_and\_pd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

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

**Instruction Operand Encoding**

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

**Description**

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

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

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

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

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

**Operation****VANDPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

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

THEN

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

ELSE

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

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**VANDPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE AND SRC2[159:128]

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

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

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

DEST[MAX\_VL-1:256] ← 0;

**VANDPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0;

**ANDPS (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]

DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]

DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]

DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VANDPS \_\_m512 \_\_mm512\_and\_ps (\_\_m512 a, \_\_m512 b);

VANDPS \_\_m512 \_\_mm512\_mask\_and\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VANDPS \_\_m512 \_\_mm512\_maskz\_and\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VANDPS \_\_m256 \_\_mm256\_mask\_and\_ps (\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);

VANDPS \_\_m256 \_\_mm256\_maskz\_and\_ps (\_\_mmask8 k, \_\_m256 a, \_\_m256 b);

VANDPS \_\_m128 \_\_mm\_mask\_and\_ps (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VANDPS \_\_m128 \_\_mm\_maskz\_and\_ps (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VANDPS \_\_m256 \_\_mm256\_and\_ps (\_\_m256 a, \_\_m256 b);

ANDPS \_\_m128 \_\_mm\_and\_ps (\_\_m128 a, \_\_m128 b);

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



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

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

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

**Operation****VANDNPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

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

THEN

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

ELSE

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

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] = 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VANDNPD (VEX.256 encoded version)**

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]

DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]

DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]

DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VANDNPD (VEX.128 encoded version)**

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]

DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

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

DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]

DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VANDNPD \_\_m512d \_mm512\_andnot\_pd (\_\_m512d a, \_\_m512d b);

VANDNPD \_\_m512d \_mm512\_mask\_andnot\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDNPD \_\_m512d \_mm512\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDNPD \_\_m256d \_mm256\_mask\_andnot\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDNPD \_\_m256d \_mm256\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDNPD \_\_m128d \_mm\_mask\_andnot\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDNPD \_\_m128d \_mm\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDNPD \_\_m256d \_mm256\_andnot\_pd (\_\_m256d a, \_\_m256d b);

ANDNPD \_\_m128d \_mm\_andnot\_pd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| OF 55 /r<br>ANDNPS xmm1, xmm2/m128  | RM         | V/V                          | SSE                      | Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem.                              |
| VEX.NDS.128.OF 55 /r<br>VANDNPS xmm1, xmm2,<br>xmm3/m128                    | RVM        | V/V                          | AVX                      | Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem.                              |
| VEX.NDS.256.OF 55 /r<br>VANDNPS ymm1, ymm2,<br>ymm3/m256                    | RVM        | V/V                          | AVX                      | Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem.                              |
| EVEX.NDS.128.OF.W0 54 /r<br>VANDPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1. |
| EVEX.NDS.256.OF.W0 54 /r<br>VANDPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1. |
| EVEX.NDS.512.OF.W0 54 /r<br>VANDPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512DQ                 | Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1. |

**Instruction Operand Encoding**

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

**Description**

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

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

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

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

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

**Operation****VANDNPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

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

THEN

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

ELSE

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

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VANDNPS (VEX.256 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]

DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]

DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]

DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]

DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]

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

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

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

DEST[MAX\_VL-1:256] ← 0

**VANDNPS (VEX.128 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]

DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]

DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]

DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0

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

DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]

DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]

DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]

DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VANDNPS \_\_m512 \_\_mm512\_andnot\_ps (\_\_m512 a, \_\_m512 b);

VANDNPS \_\_m512 \_\_mm512\_mask\_andnot\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VANDNPS \_\_m512 \_\_mm512\_maskz\_andnot\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VANDNPS \_\_m256 \_\_mm256\_mask\_andnot\_ps (\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);

VANDNPS \_\_m256 \_\_mm256\_maskz\_andnot\_ps (\_\_mmask8 k, \_\_m256 a, \_\_m256 b);

VANDNPS \_\_m128 \_\_mm\_mask\_andnot\_ps (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VANDNPS \_\_m128 \_\_mm\_maskz\_andnot\_ps (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VANDNPS \_\_m256 \_\_mm256\_andnot\_ps (\_\_m256 a, \_\_m256 b);

ANDNPS \_\_m128 \_\_mm\_andnot\_ps (\_\_m128 a, \_\_m128 b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## VBROADCAST—Load with Broadcast Floating-Point Data

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit<br>Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|---------------------------------|--------------------------|--|
| VEX.128.66.0F38.W0 18 /r<br>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<br>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<br>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<br>VBROADCASTF128 ymm1, m128                  | RM        | V/V                             | AVX                      | Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.                               |
| EVEX.256.66.0F38.W1 19 /r<br>VBROADCASTSD xmm1 {k1}{z},<br>xmm2/m64    | T1S       | V/V                             | AVX512VL<br>AVX512F      | Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.  |
| EVEX.512.66.0F38.W1 19 /r<br>VBROADCASTSD zmm1 {k1}{z},<br>xmm2/m64    | T1S       | V/V                             | AVX512F                  | Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 19 /r<br>VBROADCASTF32X2 ymm1 {k1}{z},<br>xmm2/m64 | T2        | V/V                             | AVX512VL<br>AVX512DQ     | Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.      |
| EVEX.512.66.0F38.W0 19 /r<br>VBROADCASTF32X2 zmm1 {k1}{z},<br>xmm2/m64 | T2        | V/V                             | AVX512DQ                 | Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.      |
| EVEX.128.66.0F38.W0 18 /r<br>VBROADCASTSS xmm1 {k1}{z},<br>xmm2/m32    | T1S       | V/V                             | AVX512VL<br>AVX512F      | Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.   |
| EVEX.256.66.0F38.W0 18 /r<br>VBROADCASTSS ymm1 {k1}{z},<br>xmm2/m32    | T1S       | V/V                             | AVX512VL<br>AVX512F      | Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.   |
| EVEX.512.66.0F38.W0 18 /r<br>VBROADCASTSS zmm1 {k1}{z},<br>xmm2/m32    | T1S       | V/V                             | AVX512F                  | Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.   |
| EVEX.256.66.0F38.W0 1A /r<br>VBROADCASTF32X4 ymm1 {k1}{z},<br>m128     | T4        | V/V                             | AVX512VL<br>AVX512F      | Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.     |
| EVEX.512.66.0F38.W0 1A /r<br>VBROADCASTF32X4 zmm1 {k1}{z},<br>m128     | T4        | V/V                             | AVX512F                  | Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.     |

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit<br>Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|---------------------------------|--------------------------|--|
| EVEX.256.66.0F38.W1 1A /r<br>VBROADCASTF64X2 ymm1 {k1}{z},<br>m128 | T2        | V/V                             | AVX512VL<br>AVX512DQ     | Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 1A /r<br>VBROADCASTF64X2 zmm1 {k1}{z},<br>m128 | T2        | V/V                             | AVX512DQ                 | Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in zmm1 using writemask k1. |
| EVEX.512.66.0F38.W0 1B /r<br>VBROADCASTF32X8 zmm1 {k1}{z},<br>m256 | T8        | V/V                             | AVX512DQ                 | Broadcast 256 bits of 8 single-precision floating-point data in mem to locations in zmm1 using writemask k1. |
| EVEX.512.66.0F38.W1 1B /r<br>VBROADCASTF64X4 zmm1 {k1}{z},<br>m256 | T4        | V/V                             | AVX512F                  | Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En           | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-----------------|---------------|---------------|-----------|-----------|
| RM              | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1S, T2, T4, T8 | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

#### Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

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

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.



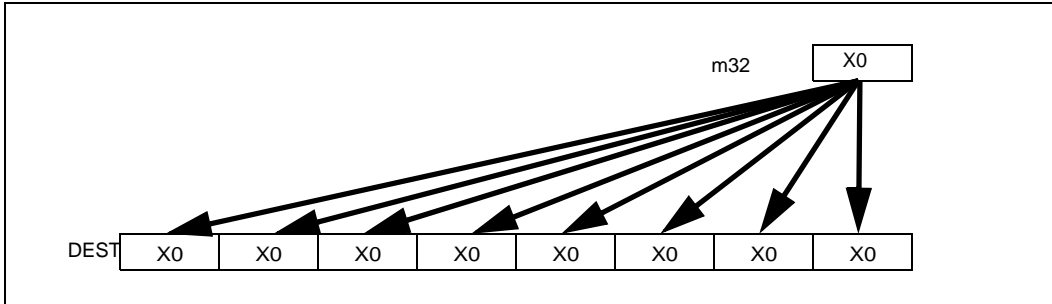


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

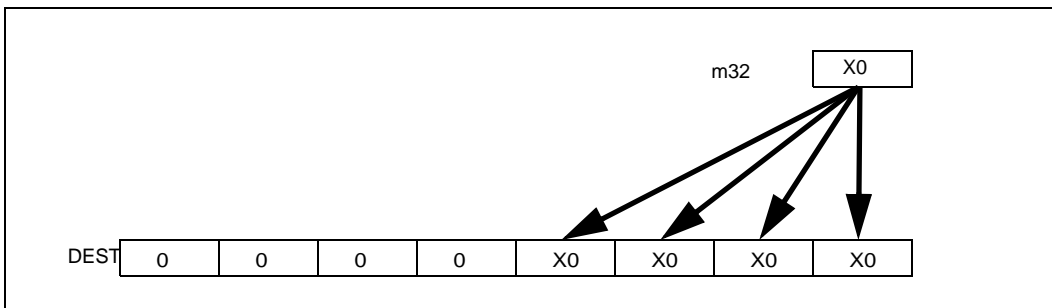


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

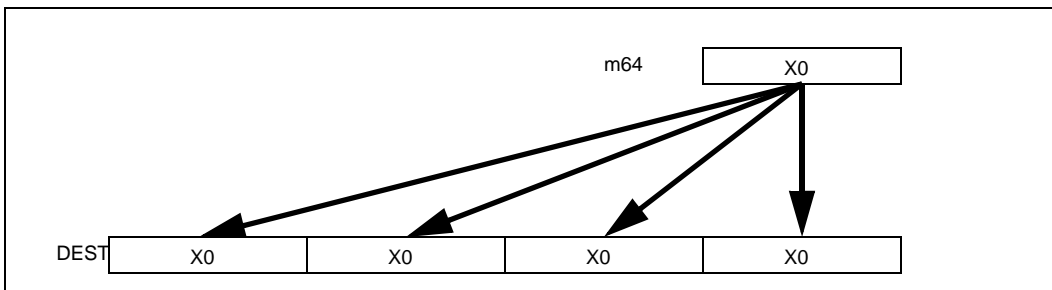


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

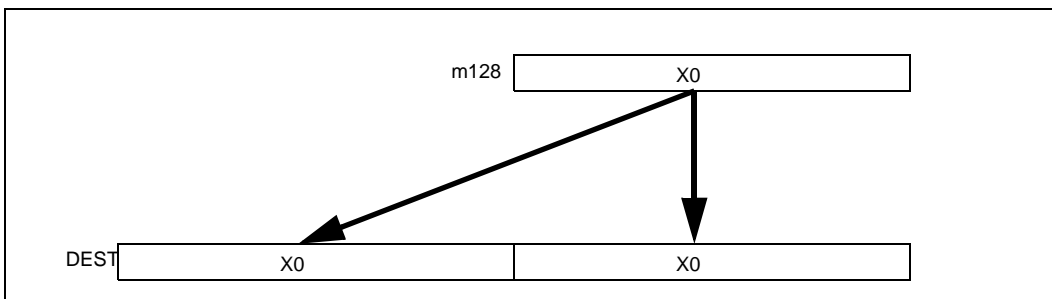


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

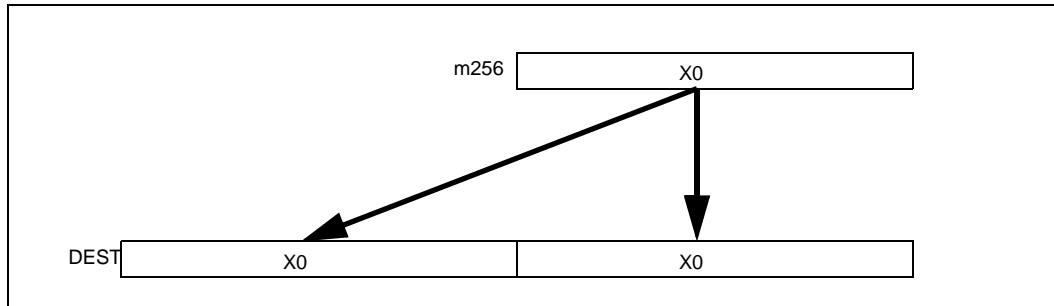


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

### Operation

#### VBROADCASTSS (128 bit version VEX and legacy)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAX_VL-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
DEST[MAX_VL-1:256] ← 0
```

#### VBROADCASTSS (EVEX encoded versions)

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

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### 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
DEST[MAX_VL-1:256] ← 0

```

**VBROADCASTSD (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[63:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF32x2 (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  n ← (j mod 2) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF128 (VEX.256 encoded version)**

```

temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAX_VL-1:256] ← 0

```

**VBROADCASTF32X4 (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  n ← (j modulo 4) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*

```

```

        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF64X2 (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    n ← (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+63:i] = 0
        FI
    FI;
ENDFOR;

```

**VBROADCASTF32X8 (EVEX.U1.512 encoded version)**

```

FOR j ← 0 TO 15
    i ← j * 32
    n ← (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF64X4 (EVEX.512 encoded version)**

```

FOR j ← 0 TO 7
    i ← j * 64
    n ← (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VBROADCASTF32x2 __m512 __mm512_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m512 __mm512_mask_broadcast_f32x2( __m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 __mm512_maskz_broadcast_f32x2( __mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 __mm256_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m256 __mm256_mask_broadcast_f32x2( __m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 __mm256_maskz_broadcast_f32x2( __mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 __mm512_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m512 __mm512_mask_broadcast_f32x4( __m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 __mm512_maskz_broadcast_f32x4( __mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 __mm256_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m256 __mm256_mask_broadcast_f32x4( __m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 __mm256_maskz_broadcast_f32x4( __mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 __mm512_broadcast_f32x8( __m256 a);
VBROADCASTF32x8 __m512 __mm512_mask_broadcast_f32x8( __m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 __mm512_maskz_broadcast_f32x8( __mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d __mm512_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m512d __mm512_mask_broadcast_f64x2( __m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d __mm512_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d __mm256_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m256d __mm256_mask_broadcast_f64x2( __m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d __mm256_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d __mm512_broadcast_f64x4( __m256d a);
VBROADCASTF64x4 __m512d __mm512_mask_broadcast_f64x4( __m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d __mm512_maskz_broadcast_f64x4( __mmask8 k, __m256d a);
VBROADCASTSD __m512d __mm512_broadcastsd_pd( __m128d a);
VBROADCASTSD __m512d __mm512_mask_broadcastsd_pd( __m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d __mm512_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d __mm256_broadcastsd_pd( __m128d a);
VBROADCASTSD __m256d __mm256_mask_broadcastsd_pd( __m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d __mm256_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d __mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 __mm512_broadcastss_ps( __m128 a);
VBROADCASTSS __m512 __mm512_mask_broadcastss_ps( __m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 __mm512_maskz_broadcastss_ps( __mmask16 k, __m128 a);
VBROADCASTSS __m256 __mm256_broadcastss_ps( __m128 a);
VBROADCASTSS __m256 __mm256_mask_broadcast_ss( __m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 __mm256_maskz_broadcast_ss( __mmask8 k, __m128 a);
VBROADCASTSS __m128 __mm_broadcastss_ps( __m128 a);
VBROADCASTSS __m128 __mm_mask_broadcast_ss( __m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 __mm_maskz_broadcast_ss( __mmask8 k, __m128 a);
VBROADCASTSS __m128 __mm_broadcast_ss(float *a);
VBROADCASTSS __m256 __mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 __mm256_broadcast_ps( __m128 * a);
VBROADCASTF128 __m256d __mm256_broadcast_pd( __m128d * a);

```

### Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6.

#UD                    If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.  
                           If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.  
                           If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

## VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit<br>Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|---------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 7A /r<br>VPBROADCASTB xmm1 {k1}{z}, reg | T1S       | V/V                             | AVX512VL<br>AVX512BW     | Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.        |
| EVEX.256.66.0F38.W0 7A /r<br>VPBROADCASTB ymm1 {k1}{z}, reg | T1S       | V/V                             | AVX512VL<br>AVX512BW     | Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.        |
| EVEX.512.66.0F38.W0 7A /r<br>VPBROADCASTB zmm1 {k1}{z}, reg | T1S       | V/V                             | AVX512BW                 | Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.        |
| EVEX.128.66.0F38.W0 7B /r<br>VPBROADCASTW xmm1 {k1}{z}, reg | T1S       | V/V                             | AVX512VL<br>AVX512BW     | Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.        |
| EVEX.256.66.0F38.W0 7B /r<br>VPBROADCASTW ymm1 {k1}{z}, reg | T1S       | V/V                             | AVX512VL<br>AVX512BW     | Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.        |
| EVEX.512.66.0F38.W0 7B /r<br>VPBROADCASTW zmm1 {k1}{z}, reg | T1S       | V/V                             | AVX512BW                 | Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.        |
| EVEX.128.66.0F38.W0 7C /r<br>VPBROADCASTD xmm1 {k1}{z}, r32 | T1S       | V/V                             | AVX512VL<br>AVX512F      | Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1. |
| EVEX.256.66.0F38.W0 7C /r<br>VPBROADCASTD ymm1 {k1}{z}, r32 | T1S       | V/V                             | AVX512VL<br>AVX512F      | Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1. |
| EVEX.512.66.0F38.W0 7C /r<br>VPBROADCASTD zmm1 {k1}{z}, r32 | T1S       | V/V                             | AVX512F                  | Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1. |
| EVEX.128.66.0F38.W1 7C /r<br>VPBROADCASTQ xmm1 {k1}{z}, r64 | T1S       | V/N.E. <sup>1</sup>             | AVX512VL<br>AVX512F      | Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.   |
| EVEX.256.66.0F38.W1 7C /r<br>VPBROADCASTQ ymm1 {k1}{z}, r64 | T1S       | V/N.E. <sup>1</sup>             | AVX512VL<br>AVX512F      | Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.   |
| EVEX.512.66.0F38.W1 7C /r<br>VPBROADCASTQ zmm1 {k1}{z}, r64 | T1S       | V/N.E. <sup>1</sup>             | AVX512F                  | Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.   |

### NOTES:

1. EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

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

### Operation

#### VPBROADCASTB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC[7:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VPBROADCASTW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC[15:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VPBROADCASTD (EVEX encoded versions)

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

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[31:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

**VPBROADCASTQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBROADCASTB \_\_m512i \_\_mm512\_mask\_set1\_epi8(\_\_m512i s, \_\_mmask64 k, int a);

VPBROADCASTB \_\_m512i \_\_mm512\_maskz\_set1\_epi8( \_\_mmask64 k, int a);

VPBROADCASTB \_\_m256i \_\_mm256\_mask\_set1\_epi8(\_\_m256i s, \_\_mmask32 k, int a);

VPBROADCASTB \_\_m256i \_\_mm256\_maskz\_set1\_epi8( \_\_mmask32 k, int a);

VPBROADCASTB \_\_m128i \_\_mm\_mask\_set1\_epi8(\_\_m128i s, \_\_mmask16 k, int a);

VPBROADCASTB \_\_m128i \_\_mm\_maskz\_set1\_epi8( \_\_mmask16 k, int a);

VPBROADCASTD \_\_m512i \_\_mm512\_mask\_set1\_epi32(\_\_m512i s, \_\_mmask16 k, int a);

VPBROADCASTD \_\_m512i \_\_mm512\_maskz\_set1\_epi32( \_\_mmask16 k, int a);

VPBROADCASTD \_\_m256i \_\_mm256\_mask\_set1\_epi32(\_\_m256i s, \_\_mmask8 k, int a);

VPBROADCASTD \_\_m256i \_\_mm256\_maskz\_set1\_epi32( \_\_mmask8 k, int a);

VPBROADCASTD \_\_m128i \_\_mm\_mask\_set1\_epi32(\_\_m128i s, \_\_mmask8 k, int a);

VPBROADCASTD \_\_m128i \_\_mm\_maskz\_set1\_epi32( \_\_mmask8 k, int a);

VPBROADCASTQ \_\_m512i \_\_mm512\_mask\_set1\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m512i \_\_mm512\_maskz\_set1\_epi64( \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m256i \_\_mm256\_mask\_set1\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m256i \_\_mm256\_maskz\_set1\_epi64( \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m128i \_\_mm\_mask\_set1\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m128i \_\_mm\_maskz\_set1\_epi64( \_\_mmask8 k, \_\_int64 a);

VPBROADCASTW \_\_m512i \_\_mm512\_mask\_set1\_epi16(\_\_m512i s, \_\_mmask32 k, int a);

VPBROADCASTW \_\_m512i \_\_mm512\_maskz\_set1\_epi16( \_\_mmask32 k, int a);

VPBROADCASTW \_\_m256i \_\_mm256\_mask\_set1\_epi16(\_\_m256i s, \_\_mmask16 k, int a);

VPBROADCASTW \_\_m256i \_\_mm256\_maskz\_set1\_epi16( \_\_mmask16 k, int a);

VPBROADCASTW \_\_m128i \_\_mm\_mask\_set1\_epi16(\_\_m128i s, \_\_mmask8 k, int a);

VPBROADCASTW \_\_m128i \_\_mm\_maskz\_set1\_epi16( \_\_mmask8 k, int a);

**Exceptions**

EVEX-encoded instructions, see Exceptions Type E7NM.

#UD If EVEX.vvvv != 1111B.



## VPBROADCAST—Load Integer and Broadcast

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 78 /r<br>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<br>VPBROADCASTB ymm1, xmm2/m8           | RM         | V/V                          | AVX2                     | Broadcast a byte integer in the source operand to thirty-two locations in ymm1.                 |
| EVEX.128.66.0F38.W0 78 /r<br>VPBROADCASTB xmm1{k1}{z}, xmm2/m8   | T1S        | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.W0 78 /r<br>VPBROADCASTB ymm1{k1}{z}, xmm2/m8   | T1S        | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.    |
| EVEX.512.66.0F38.W0 78 /r<br>VPBROADCASTB zmm1{k1}{z}, xmm2/m8   | T1S        | V/V                          | AVX512BW                 | Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1. |
| VEX.128.66.0F38.W0 79 /r<br>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<br>VPBROADCASTW ymm1, xmm2/m16          | RM         | V/V                          | AVX2                     | Broadcast a word integer in the source operand to sixteen locations in ymm1.                    |
| EVEX.128.66.0F38.W0 79 /r<br>VPBROADCASTW xmm1{k1}{z}, xmm2/m16  | T1S        | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.W0 79 /r<br>VPBROADCASTW ymm1{k1}{z}, xmm2/m16  | T1S        | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.    |
| EVEX.512.66.0F38.W0 79 /r<br>VPBROADCASTW zmm1{k1}{z}, xmm2/m16  | T1S        | V/V                          | AVX512BW                 | Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1. |
| VEX.128.66.0F38.W0 58 /r<br>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<br>VPBROADCASTD ymm1, xmm2/m32          | RM         | V/V                          | AVX2                     | Broadcast a dword integer in the source operand to eight locations in ymm1.                     |
| EVEX.128.66.0F38.W0 58 /r<br>VPBROADCASTD xmm1 {k1}{z}, xmm2/m32 | T1S        | V/V                          | AVX512VL<br>AVX512F      | Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.   |
| EVEX.256.66.0F38.W0 58 /r<br>VPBROADCASTD ymm1 {k1}{z}, xmm2/m32 | T1S        | V/V                          | AVX512VL<br>AVX512F      | Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.   |
| EVEX.512.66.0F38.W0 58 /r<br>VPBROADCASTD zmm1 {k1}{z}, xmm2/m32 | T1S        | V/V                          | AVX512F                  | Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.   |
| VEX.128.66.0F38.W0 59 /r<br>VPBROADCASTQ xmm1, xmm2/m64          | RM         | V/V                          | AVX2                     | Broadcast a qword element in source operand to two locations in xmm1.                           |

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.256.66.0F38.W0 59 /r<br>VPBROADCASTQ ymm1, xmm2/m64             | RM        | V/V                          | AVX2                     | Broadcast a qword element in source operand to four locations in ymm1.                          |
| EVEX.128.66.0F38.W1 59 /r<br>VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64    | T1S       | V/V                          | AVX512VL<br>AVX512F      | Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.       |
| EVEX.256.66.0F38.W1 59 /r<br>VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64    | T1S       | V/V                          | AVX512VL<br>AVX512F      | Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.       |
| EVEX.512.66.0F38.W1 59 /r<br>VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64    | T1S       | V/V                          | AVX512F                  | Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.       |
| EVEX.128.66.0F38.W0 59 /r<br>VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64 | T2        | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.W0 59 /r<br>VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64 | T2        | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.    |
| EVEX.512.66.0F38.W0 59 /r<br>VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64 | T2        | V/V                          | AVX512DQ                 | Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.    |
| VEX.256.66.0F38.W0 5A /r<br>VBROADCASTI128 ymm1, m128               | RM        | V/V                          | AVX2                     | Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.                     |
| EVEX.256.66.0F38.W0 5A /r<br>VBROADCASTI32X4 ymm1 {k1}{z}, m128     | T4        | V/V                          | AVX512VL<br>AVX512F      | Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 5A /r<br>VBROADCASTI32X4 zmm1 {k1}{z}, m128     | T4        | V/V                          | AVX512F                  | Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 5A /r<br>VBROADCASTI64X2 ymm1 {k1}{z}, m128     | T2        | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.   |
| EVEX.512.66.0F38.W1 5A /r<br>VBROADCASTI64X2 zmm1 {k1}{z}, m128     | T2        | V/V                          | AVX512DQ                 | Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.   |
| EVEX.512.66.0F38.W1 5B /r<br>VBROADCASTI32X8 zmm1 {k1}{z}, m256     | T8        | V/V                          | AVX512DQ                 | Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1. |
| EVEX.512.66.0F38.W1 5B /r<br>VBROADCASTI64X4 zmm1 {k1}{z}, m256     | T4        | V/V                          | AVX512F                  | Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.   |

## Instruction Operand Encoding

| Op/En           | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-----------------|---------------|---------------|-----------|-----------|
| RM              | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1S, T2, T4, T8 | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

## Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: 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. The destination operand is a YMM register.

VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

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

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

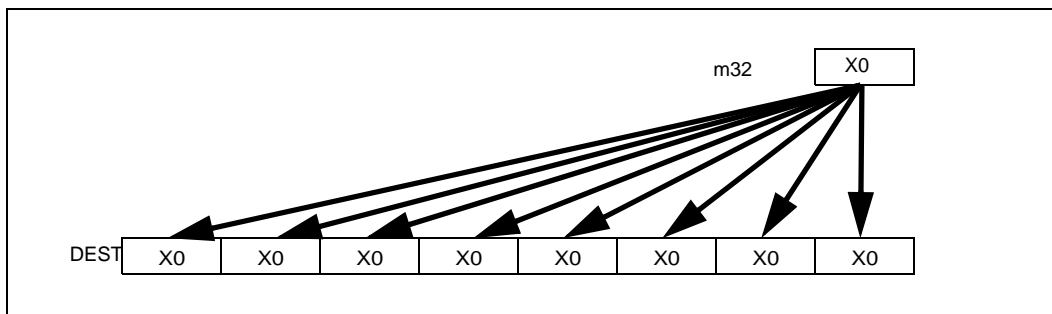


Figure 5-6. VPBROADCASTD Operation (VEX.256 encoded version)

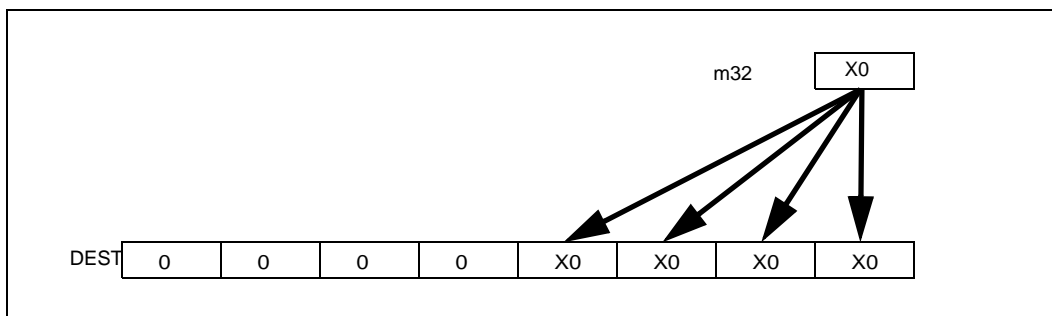


Figure 5-7. VPBROADCASTD Operation (128-bit version)

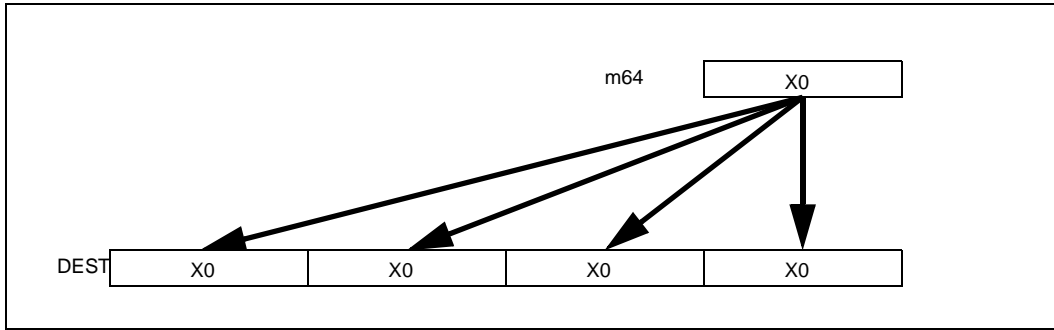


Figure 5-8. VPBROADCASTQ Operation (256-bit version)

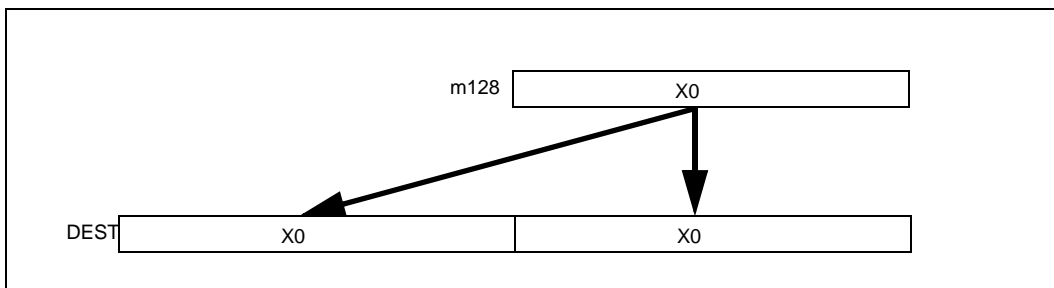


Figure 5-9. VBROADCASTI128 Operation (256-bit version)

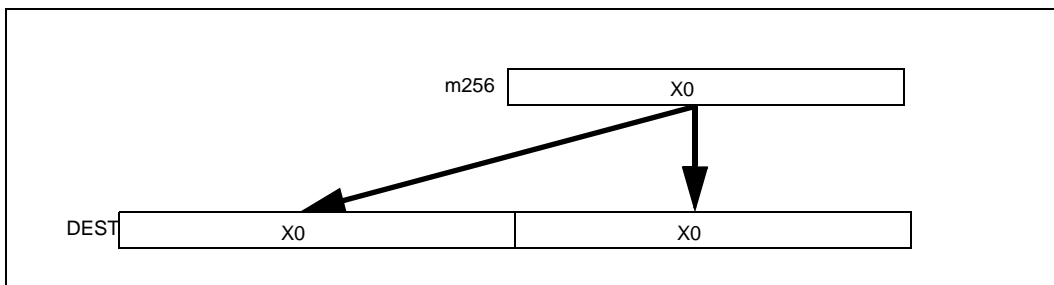


Figure 5-10. VBROADCASTI256 Operation (512-bit version)

**Operation**

**VPBROADCASTB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SRC[7:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+7:i] ← 0

  FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPBROADCASTW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPBROADCASTD (128 bit version)**

```

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

```

**VPBROADCASTD (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
DEST[MAX_VL-1:256] ← 0

```

**VPBROADCASTD (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

DEST[MAX\_VL-1:VL] ← 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  
 DEST[MAX\_VL-1:256] ← 0

**VPBROADCASTQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ← SRC[63:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VBROADCASTI32x2 (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  n ← (j mod 2) \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[n+31:n]

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VBROADCASTI128 (VEX.256 encoded version)**

temp ← SRC[127:0]  
 DEST[127:0] ← temp  
 DEST[255:128] ← temp  
 DEST[MAX\_VL-1:256] ← 0

**VBROADCASTI32x4 (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  n ← (j modulo 4) \* 32

  IF k1[j] OR \*no writemask\*

```

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

```

**VBROADCASTI64X2 (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  n ← (j modulo 2) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] = 0
    FI
  FI;
ENDFOR;

```

**VBROADCASTI32X8 (EVEX.U1.512 encoded version)**

```

FOR j ← 0 TO 15
  i ← j * 32
  n ← (j modulo 8) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTI64X4 (EVEX.512 encoded version)**

```

FOR j ← 0 TO 7
  i ← j * 64
  n ← (j modulo 4) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;

```

FI

FI;  
 ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB \_\_m512i \_\_mm512\_broadcastb\_epi8( \_\_m128i a);  
 VPBROADCASTB \_\_m512i \_\_mm512\_mask\_broadcastb\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m128i a);  
 VPBROADCASTB \_\_m512i \_\_mm512\_maskz\_broadcastb\_epi8( \_\_mmask64 k, \_\_m128i a);  
 VPBROADCASTB \_\_m256i \_\_mm256\_broadcastb\_epi8(\_\_m128i a);  
 VPBROADCASTB \_\_m256i \_\_mm256\_mask\_broadcastb\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m128i a);  
 VPBROADCASTB \_\_m256i \_\_mm256\_maskz\_broadcastb\_epi8( \_\_mmask32 k, \_\_m128i a);  
 VPBROADCASTB \_\_m128i \_\_mm\_mask\_broadcastb\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a);  
 VPBROADCASTB \_\_m128i \_\_mm\_maskz\_broadcastb\_epi8( \_\_mmask16 k, \_\_m128i a);  
 VPBROADCASTB \_\_m128i \_\_mm\_broadcastb\_epi8(\_\_m128i a);  
 VPBROADCASTD \_\_m512i \_\_mm512\_broadcastd\_epi32( \_\_m128i a);  
 VPBROADCASTD \_\_m512i \_\_mm512\_mask\_broadcastd\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VPBROADCASTD \_\_m512i \_\_mm512\_maskz\_broadcastd\_epi32( \_\_mmask16 k, \_\_m128i a);  
 VPBROADCASTD \_\_m256i \_\_mm256\_broadcastd\_epi32( \_\_m128i a);  
 VPBROADCASTD \_\_m256i \_\_mm256\_mask\_broadcastd\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTD \_\_m256i \_\_mm256\_maskz\_broadcastd\_epi32( \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTD \_\_m128i \_\_mm\_broadcastd\_epi32(\_\_m128i a);  
 VPBROADCASTD \_\_m128i \_\_mm\_mask\_broadcastd\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTD \_\_m128i \_\_mm\_maskz\_broadcastd\_epi32( \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTQ \_\_m512i \_\_mm512\_broadcastq\_epi64( \_\_m128i a);  
 VPBROADCASTQ \_\_m512i \_\_mm512\_mask\_broadcastq\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTQ \_\_m512i \_\_mm512\_maskz\_broadcastq\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTQ \_\_m256i \_\_mm256\_broadcastq\_epi64(\_\_m128i a);  
 VPBROADCASTQ \_\_m256i \_\_mm256\_mask\_broadcastq\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTQ \_\_m256i \_\_mm256\_maskz\_broadcastq\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTQ \_\_m128i \_\_mm\_broadcastq\_epi64(\_\_m128i a);  
 VPBROADCASTQ \_\_m128i \_\_mm\_mask\_broadcastq\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTQ \_\_m128i \_\_mm\_maskz\_broadcastq\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTW \_\_m512i \_\_mm512\_broadcastw\_epi16(\_\_m128i a);  
 VPBROADCASTW \_\_m512i \_\_mm512\_mask\_broadcastw\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m128i a);  
 VPBROADCASTW \_\_m512i \_\_mm512\_maskz\_broadcastw\_epi16( \_\_mmask32 k, \_\_m128i a);  
 VPBROADCASTW \_\_m256i \_\_mm256\_broadcastw\_epi16(\_\_m128i a);  
 VPBROADCASTW \_\_m256i \_\_mm256\_mask\_broadcastw\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m128i a);  
 VPBROADCASTW \_\_m256i \_\_mm256\_maskz\_broadcastw\_epi16( \_\_mmask16 k, \_\_m128i a);  
 VPBROADCASTW \_\_m128i \_\_mm\_broadcastw\_epi16(\_\_m128i a);  
 VPBROADCASTW \_\_m128i \_\_mm\_mask\_broadcastw\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);  
 VPBROADCASTW \_\_m128i \_\_mm\_maskz\_broadcastw\_epi16( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m512i \_\_mm512\_broadcast\_i32x2( \_\_m128i a);  
 VBROADCASTI32x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x2(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x2( \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_broadcast\_i32x2( \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x2( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_broadcastq\_i32x2(\_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_mask\_broadcastq\_i32x2(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_maskz\_broadcastq\_i32x2( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_broadcast\_i32x4( \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x4(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x4( \_\_mmask16 k, \_\_m128i a);



VBROADCASTI32x4 \_\_m256i \_\_mm256\_broadcast\_i32x4( \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x4(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x4( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_broadcast\_i32x8( \_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x8(\_\_m512i s, \_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x8( \_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_broadcast\_i64x2( \_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x2(\_\_m512i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x2( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_broadcast\_i64x2( \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i64x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i64x2( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_broadcast\_i64x4( \_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x4(\_\_m512i s, \_\_mmask8 k, \_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x4( \_\_mmask8 k, \_\_m256i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD                    If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.  
                           If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.  
                           If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

## CMPPD—Compare Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F C2 /r ib<br>CMPPD xmm1, xmm2/m128, imm8  | RMI        | V/V                          | SSE2                     | Compare packed double-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.  |
| VEX.NDS.128.66.0F.WIG C2 /r ib<br>VCMPD xmm1, xmm2, xmm3/m128,<br>imm8                 | RVMI       | V/V                          | AVX                      | Compare packed double-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.  |
| VEX.NDS.256.66.0F.WIG C2 /r ib<br>VCMPD ymm1, ymm2, ymm3/m256,<br>imm8                 | RVMI       | V/V                          | AVX                      | Compare packed double-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.  |
| EVEX.NDS.128.66.0F.W1 C2 /r ib<br>VCMPD k1 {k2}, xmm2,<br>xmm3/m128/m64bcst, imm8      | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare packed double-precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.256.66.0F.W1 C2 /r ib<br>VCMPD k1 {k2}, ymm2,<br>ymm3/m256/m64bcst, imm8      | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare packed double-precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.512.66.0F.W1 C2 /r ib<br>VCMPD k1 {k2}, zmm2,<br>zmm3/m512/m64bcst{sae}, imm8 | FV         | V/V                          | AVX512F                  | Compare packed double-precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | Imm8      |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | Imm8      |

### Description

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

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

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with

results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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

The comparison predicate operand is an 8-bit immediate:

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

**Table 5-3. Comparison Predicate for CMPPD and CMPPS Instructions**

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

**Table 5-3. Comparison Predicate for CMPPD and CMPPS Instructions (Continued)**

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

**NOTES:**

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

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

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

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 5-4. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 5-4. Pseudo-Op and CMPPD Implementation**

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

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

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 5-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 5-5, where the notations of reg1, reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 5-5.

**Table 5-5. Pseudo-Op and VCMPPD Implementation**

| <b>Pseudo-Op</b>                      | <b>CMPPD Implementation</b>         |
|---------------------------------------|-------------------------------------|
| VCMPEQPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 0</i>   |
| VCMPLTPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 1</i>   |
| VCMLEPD <i>reg1, reg2, reg3</i>       | VCMPPD <i>reg1, reg2, reg3, 2</i>   |
| VCMUNORDPD <i>reg1, reg2, reg3</i>    | VCMPPD <i>reg1, reg2, reg3, 3</i>   |
| VCMNEQPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 4</i>   |
| VCMNLTPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 5</i>   |
| VCMNLEPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 6</i>   |
| VCMORDPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 7</i>   |
| VCMPEQ_UQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 8</i>   |
| VCMNGEPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 9</i>   |
| VCMNGTPD <i>reg1, reg2, reg3</i>      | VCMPPD <i>reg1, reg2, reg3, 0AH</i> |
| VCMFALSEPD <i>reg1, reg2, reg3</i>    | VCMPPD <i>reg1, reg2, reg3, 0BH</i> |
| VCMNEQ_OQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 0CH</i> |
| VCMGEPD <i>reg1, reg2, reg3</i>       | VCMPPD <i>reg1, reg2, reg3, 0DH</i> |
| VCMGTPD <i>reg1, reg2, reg3</i>       | VCMPPD <i>reg1, reg2, reg3, 0EH</i> |
| VCMTRUEPD <i>reg1, reg2, reg3</i>     | VCMPPD <i>reg1, reg2, reg3, 0FH</i> |
| VCMPEQ_OSPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 10H</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 11H</i> |
| VCMLE_OQPD <i>reg1, reg2, reg3</i>    | VCMPPD <i>reg1, reg2, reg3, 12H</i> |
| VCMUNORD_SPD <i>reg1, reg2, reg3</i>  | VCMPPD <i>reg1, reg2, reg3, 13H</i> |
| VCMNEQ_USPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 14H</i> |
| VCMNLT_UQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 15H</i> |
| VCMNLE_UQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 16H</i> |
| VCMORD_SPD <i>reg1, reg2, reg3</i>    | VCMPPD <i>reg1, reg2, reg3, 17H</i> |
| VCMPEQ_USPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 18H</i> |
| VCMNGE_UQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 19H</i> |
| VCMNGT_UQPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 1AH</i> |
| VCMFALSE_OSPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 1BH</i> |
| VCMNEQ_OSPD <i>reg1, reg2, reg3</i>   | VCMPPD <i>reg1, reg2, reg3, 1CH</i> |
| VCMGE_OQPD <i>reg1, reg2, reg3</i>    | VCMPPD <i>reg1, reg2, reg3, 1DH</i> |
| VCMGT_OQPD <i>reg1, reg2, reg3</i>    | VCMPPD <i>reg1, reg2, reg3, 1EH</i> |
| VCMTRUE_USPD <i>reg1, reg2, reg3</i>  | VCMPPD <i>reg1, reg2, reg3, 1FH</i> |

**Operation**

CASE (COMPARISON PREDICATE) OF

```

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

```

ESAC;

**VCMPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

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

THEN

CMP ← SRC1[i+63:i] OP5 SRC2[63:0]

ELSE

CMP ← SRC1[i+63:i] OP5 SRC2[i+63:i]

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

```
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VCMPD (VEX.256 encoded version)**

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
CMP1 ← SRC1[127:64] OP5 SRC2[127:64];
CMP2 ← SRC1[191:128] OP5 SRC2[191:128];
CMP3 ← SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[191:128] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[191:128] ← 0000000000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[255:192] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[255:192] ← 0000000000000000H; FI;
DEST[MAX_VL-1:256] ← 0
```

**VCMPD (VEX.128 encoded version)**

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

**CMPPD (128-bit Legacy SSE version)**

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

**Intel C/C++ Compiler Intrinsic Equivalent**

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

(V)CMPPD \_\_m128 \_\_mm\_cmp\_pd(\_\_m128d a, \_\_m128d b, int imm)

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-3.

Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## CMPPS—Compare Packed Single-Precision Floating-Point Values

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

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | Imm8      |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | Imm8      |

### Description

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

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

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with

results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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

The comparison predicate operand is an 8-bit immediate:

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

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

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

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 5-6. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 5-6. Pseudo-Op and CMPPS Implementation**

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

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 5-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 5-7, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 5-7.

Table 5-7. Pseudo-Op and VCMPPS Implementation

| Pseudo-Op                               | CMPPS Implementation                |
|---|-------------------------------------|
| VCMPEQPS <i>reg1, reg2, reg3</i>        | VCMPPS <i>reg1, reg2, reg3, 0</i>   |
| VCMPLTPS <i>reg1, reg2, reg3</i>        | VCMPPS <i>reg1, reg2, reg3, 1</i>   |
| VCMPLGPS <i>reg1, reg2, reg3</i>        | VCMPPS <i>reg1, reg2, reg3, 2</i>   |
| VCMPLNDPS <i>reg1, reg2, reg3</i>       | VCMPPS <i>reg1, reg2, reg3, 3</i>   |
| VCMPLNEQPS <i>reg1, reg2, reg3</i>      | VCMPPS <i>reg1, reg2, reg3, 4</i>   |
| VCMPLNLTGPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 5</i>   |
| VCMPLNLEGPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 6</i>   |
| VCMPLNDPS <i>reg1, reg2, reg3</i>       | VCMPPS <i>reg1, reg2, reg3, 7</i>   |
| VCMPEQ_UGPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 8</i>   |
| VCMPLNGGPS <i>reg1, reg2, reg3</i>      | VCMPPS <i>reg1, reg2, reg3, 9</i>   |
| VCMPLNGTGPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 0AH</i> |
| VCMPLFALSEGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 0BH</i> |
| VCMPLNEQ_OGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 0CH</i> |
| VCMPLNGGPS <i>reg1, reg2, reg3</i>      | VCMPPS <i>reg1, reg2, reg3, 0DH</i> |
| VCMPLGTGPS <i>reg1, reg2, reg3</i>      | VCMPPS <i>reg1, reg2, reg3, 0EH</i> |
| VCMPLTRUEGPS <i>reg1, reg2, reg3</i>    | VCMPPS <i>reg1, reg2, reg3, 0FH</i> |
| VCMPEQ_OGPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 10H</i> |
| VCMPLT_OGPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 11H</i> |
| VCMPLGPS_OGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 12H</i> |
| VCMPLND_SPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 13H</i> |
| VCMPLNEQ_USPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 14H</i> |
| VCMPLNLT_UGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 15H</i> |
| VCMPLNLE_UGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 16H</i> |
| VCMPLND_SPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 17H</i> |
| VCMPEQ_USPS <i>reg1, reg2, reg3</i>     | VCMPPS <i>reg1, reg2, reg3, 18H</i> |
| VCMPLNGE_UGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 19H</i> |
| VCMPLNGT_UGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 1AH</i> |
| VCMPLFALSE_OGPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 1BH</i> |
| VCMPLNEQ_OGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 1CH</i> |
| VCMPLNGE_OGPS <i>reg1, reg2, reg3</i>   | VCMPPS <i>reg1, reg2, reg3, 1DH</i> |
| VCMPLGT_OGPS <i>reg1, reg2, reg3</i>    | VCMPPS <i>reg1, reg2, reg3, 1EH</i> |
| VCMPLTRUE_USPS <i>reg1, reg2, reg3</i>  | VCMPPS <i>reg1, reg2, reg3, 1FH</i> |

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ← EQ\_OQ; OP5 ← EQ\_OQ;
- 1: OP3 ← LT\_OS; OP5 ← LT\_OS;
- 2: OP3 ← LE\_OS; OP5 ← LE\_OS;
- 3: OP3 ← UNORD\_Q; OP5 ← UNORD\_Q;
- 4: OP3 ← NEQ\_UQ; OP5 ← NEQ\_UQ;
- 5: OP3 ← NLT\_US; OP5 ← NLT\_US;

6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;  
 7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;  
 8: OP5  $\leftarrow$  EQ\_UQ;  
 9: OP5  $\leftarrow$  NGE\_US;  
 10: OP5  $\leftarrow$  NGT\_US;  
 11: OP5  $\leftarrow$  FALSE\_OQ;  
 12: OP5  $\leftarrow$  NEQ\_OQ;  
 13: OP5  $\leftarrow$  GE\_OS;  
 14: OP5  $\leftarrow$  GT\_OS;  
 15: OP5  $\leftarrow$  TRUE\_UQ;  
 16: OP5  $\leftarrow$  EQ\_OS;  
 17: OP5  $\leftarrow$  LT\_OQ;  
 18: OP5  $\leftarrow$  LE\_OQ;  
 19: OP5  $\leftarrow$  UNORD\_S;  
 20: OP5  $\leftarrow$  NEQ\_US;  
 21: OP5  $\leftarrow$  NLT\_UQ;  
 22: OP5  $\leftarrow$  NLE\_UQ;  
 23: OP5  $\leftarrow$  ORD\_S;  
 24: OP5  $\leftarrow$  EQ\_US;  
 25: OP5  $\leftarrow$  NGE\_UQ;  
 26: OP5  $\leftarrow$  NGT\_UQ;  
 27: OP5  $\leftarrow$  FALSE\_OS;  
 28: OP5  $\leftarrow$  NEQ\_OS;  
 29: OP5  $\leftarrow$  GE\_OQ;  
 30: OP5  $\leftarrow$  GT\_OQ;  
 31: OP5  $\leftarrow$  TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### VCMPPS (EVEX encoded versions)

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

FOR j  $\leftarrow$  0 TO KL-1

i  $\leftarrow$  j \* 32

IF k2[j] OR \*no writemask\*

THEN

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

THEN

CMP  $\leftarrow$  SRC1[i+31:i] OP5 SRC2[31:0]

ELSE

CMP  $\leftarrow$  SRC1[i+31:i] OP5 SRC2[j+31:i]

FI;

IF CMP = TRUE

THEN DEST[j]  $\leftarrow$  1;

ELSE DEST[j]  $\leftarrow$  0; FI;

ELSE DEST[j]  $\leftarrow$  0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL]  $\leftarrow$  0

**VCMPPS (VEX.256 encoded version)**

```

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
CMP4 ← SRC1[159:128] OP5 SRC2[159:128];
CMP5 ← SRC1[191:160] OP5 SRC2[191:160];
CMP6 ← SRC1[223:192] OP5 SRC2[223:192];
CMP7 ← SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] ← FFFFFFFFH;
    ELSE DEST[159:128] ← 00000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] ← FFFFFFFFH;
    ELSE DEST[191:160] ← 00000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] ← FFFFFFFFH;
    ELSE DEST[223:192] ← 00000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] ← FFFFFFFFH;
    ELSE DEST[255:224] ← 00000000H; FI;
DEST[MAX_VL-1:256] ← 0

```

**VCMPPS (VEX.128 encoded version)**

```

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

```

**CMPPS (128-bit Legacy SSE version)**

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCMPSS __mmask16 _mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPSS __mmask16 _mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPSS __mmask16 _mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPSS __mmask16 _mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPSS __mmask8 _mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPSS __mmask8 _mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPSS __mmask8 _mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __m256 _mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPSS __m128 _mm_cmp_ps(__m128 a, __m128 b, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-3.

Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## CMPSD—Compare Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F2 0F C2 /r ib<br>CMPSD xmm1, xmm2/m64, imm8                                   | RMI        | V/V                          | SSE2                     | Compare low double-precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.  |
| VEX.NDS.128.F2.0F.WIG C2 /r ib<br>VCMPSD xmm1, xmm2,<br>xmm3/m64, imm8         | RVMI       | V/V                          | AVX                      | Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.  |
| EVEX.NDS.LIG.F2.0F.W1 C2 /r ib<br>VCMPSD k1 {k2}, xmm2,<br>xmm3/m64{sae}, imm8 | T1S        | V/V                          | AVX512F                  | Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | Imm8      |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | Imm8      |

### Description

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

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAX\_VL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

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

The comparison predicate operand is an 8-bit immediate:

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

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

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

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either

by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 5-8. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 5-8. Pseudo-Op and CMPSD Implementation**

| Pseudo-Op                    | CMPSD Implementation       |
|------------------------------|----------------------------|
| CMPEQSD <i>xmm1, xmm2</i>    | CMPSD <i>xmm1, xmm2, 0</i> |
| CMPLTSD <i>xmm1, xmm2</i>    | CMPSD <i>xmm1, xmm2, 1</i> |
| CMPLSD <i>xmm1, xmm2</i>     | CMPSD <i>xmm1, xmm2, 2</i> |
| CMPUNORDSD <i>xmm1, xmm2</i> | CMPSD <i>xmm1, xmm2, 3</i> |
| CMPNEQSD <i>xmm1, xmm2</i>   | CMPSD <i>xmm1, xmm2, 4</i> |
| CMPNLTSD <i>xmm1, xmm2</i>   | CMPSD <i>xmm1, xmm2, 5</i> |
| CMPNLESD <i>xmm1, xmm2</i>   | CMPSD <i>xmm1, xmm2, 6</i> |
| CMPORDSD <i>xmm1, xmm2</i>   | CMPSD <i>xmm1, xmm2, 7</i> |

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

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 5-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 5-9, where the notations of *reg1*, *reg2*, and *reg3* represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 5-9.

**Table 5-9. Pseudo-Op and VCMPSD Implementation**

| Pseudo-Op                            | CMPSD Implementation                |
|--------------------------------------|-------------------------------------|
| VCMPEQSD <i>reg1, reg2, reg3</i>     | VCMPSD <i>reg1, reg2, reg3, 0</i>   |
| VCMPLTSD <i>reg1, reg2, reg3</i>     | VCMPSD <i>reg1, reg2, reg3, 1</i>   |
| VCMPLSD <i>reg1, reg2, reg3</i>      | VCMPSD <i>reg1, reg2, reg3, 2</i>   |
| VCMPUNORDSD <i>reg1, reg2, reg3</i>  | VCMPSD <i>reg1, reg2, reg3, 3</i>   |
| VCMPNEQSD <i>reg1, reg2, reg3</i>    | VCMPSD <i>reg1, reg2, reg3, 4</i>   |
| VCMPNLTSD <i>reg1, reg2, reg3</i>    | VCMPSD <i>reg1, reg2, reg3, 5</i>   |
| VCMPNLESD <i>reg1, reg2, reg3</i>    | VCMPSD <i>reg1, reg2, reg3, 6</i>   |
| VCMPORDSD <i>reg1, reg2, reg3</i>    | VCMPSD <i>reg1, reg2, reg3, 7</i>   |
| VCMPEQ_UQSD <i>reg1, reg2, reg3</i>  | VCMPSD <i>reg1, reg2, reg3, 8</i>   |
| VCMPNGESD <i>reg1, reg2, reg3</i>    | VCMPSD <i>reg1, reg2, reg3, 9</i>   |
| VCMPNGTSD <i>reg1, reg2, reg3</i>    | VCMPSD <i>reg1, reg2, reg3, 0AH</i> |
| VCMFALSESD <i>reg1, reg2, reg3</i>   | VCMPSD <i>reg1, reg2, reg3, 0BH</i> |
| VCMPNEQ_OQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 0CH</i> |
| VCMPGESD <i>reg1, reg2, reg3</i>     | VCMPSD <i>reg1, reg2, reg3, 0DH</i> |



Table 5-9. Pseudo-Op and VCMPSPD Implementation

| Pseudo-Op                              | VCMPSPD Implementation               |
|--|--------------------------------------|
| VCMPGTSD <i>reg1, reg2, reg3</i>       | VCMPSPD <i>reg1, reg2, reg3, 0EH</i> |
| VCMPTRUESD <i>reg1, reg2, reg3</i>     | VCMPSPD <i>reg1, reg2, reg3, 0FH</i> |
| VCMPPEQ_OSSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 10H</i> |
| VCMPPLT_OQSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 11H</i> |
| VCMPLE_OQSD <i>reg1, reg2, reg3</i>    | VCMPSPD <i>reg1, reg2, reg3, 12H</i> |
| VCMPUNORD_SSD <i>reg1, reg2, reg3</i>  | VCMPSPD <i>reg1, reg2, reg3, 13H</i> |
| VCMPNEQ_USSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 14H</i> |
| VCMPNLT_UQSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 15H</i> |
| VCMPNLE_UQSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 16H</i> |
| VCMPORD_SSD <i>reg1, reg2, reg3</i>    | VCMPSPD <i>reg1, reg2, reg3, 17H</i> |
| VCMPPEQ_USSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 18H</i> |
| VCMPNGE_UQSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 19H</i> |
| VCMPNGT_UQSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 1AH</i> |
| VCMPFALSE_OSSD <i>reg1, reg2, reg3</i> | VCMPSPD <i>reg1, reg2, reg3, 1BH</i> |
| VCMPNEQ_OSSD <i>reg1, reg2, reg3</i>   | VCMPSPD <i>reg1, reg2, reg3, 1CH</i> |
| VCMPGE_OQSD <i>reg1, reg2, reg3</i>    | VCMPSPD <i>reg1, reg2, reg3, 1DH</i> |
| VCMPGT_OQSD <i>reg1, reg2, reg3</i>    | VCMPSPD <i>reg1, reg2, reg3, 1EH</i> |
| VCMPTRUE_USSD <i>reg1, reg2, reg3</i>  | VCMPSPD <i>reg1, reg2, reg3, 1FH</i> |

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

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ←EQ\_OQ; OP5 ←EQ\_OQ;
- 1: OP3 ←LT\_OS; OP5 ←LT\_OS;
- 2: OP3 ←LE\_OS; OP5 ←LE\_OS;
- 3: OP3 ←UNORD\_Q; OP5 ←UNORD\_Q;
- 4: OP3 ←NEQ\_UQ; OP5 ←NEQ\_UQ;
- 5: OP3 ←NLT\_US; OP5 ←NLT\_US;
- 6: OP3 ←NLE\_US; OP5 ←NLE\_US;
- 7: OP3 ←ORD\_Q; OP5 ←ORD\_Q;
- 8: OP5 ←EQ\_UQ;
- 9: OP5 ←NGE\_US;
- 10: OP5 ←NGT\_US;
- 11: OP5 ←FALSE\_OQ;
- 12: OP5 ←NEQ\_OQ;
- 13: OP5 ←GE\_OS;
- 14: OP5 ←GT\_OS;
- 15: OP5 ←TRUE\_UQ;
- 16: OP5 ←EQ\_OS;
- 17: OP5 ←LT\_OQ;
- 18: OP5 ←LE\_OQ;
- 19: OP5 ←UNORD\_S;
- 20: OP5 ←NEQ\_US;
- 21: OP5 ←NLT\_UQ;

22: OP5 ← NLE\_UQ;  
 23: OP5 ← ORD\_S;  
 24: OP5 ← EQ\_US;  
 25: OP5 ← NGE\_UQ;  
 26: OP5 ← NGT\_UQ;  
 27: OP5 ← FALSE\_OS;  
 28: OP5 ← NEQ\_OS;  
 29: OP5 ← GE\_OQ;  
 30: OP5 ← GT\_OQ;  
 31: OP5 ← TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### VCMPSD (EVEX encoded version)

CMPO ← SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or \*no writemask\*

THEN IF CMPO = TRUE

THEN DEST[0] ← 1;

ELSE DEST[0] ← 0; FI;

ELSE DEST[0] ← 0 ; zeroing-masking only

FI;

DEST[MAX\_KL-1:1] ← 0

#### CMPSD (128-bit Legacy SSE version)

CMPO ← DEST[63:0] OP3 SRC[63:0];

IF CMPO = TRUE

THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;

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

DEST[MAX\_VL-1:64] (Unmodified)

#### VCMPSD (VEX.128 encoded version)

CMPO ← SRC1[63:0] OP5 SRC2[63:0];

IF CMPO = TRUE

THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;

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

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

VCMPSD \_\_mmask8 \_mm\_cmp\_sd\_mask( \_\_m128d a, \_\_m128d b, int imm);

VCMPSD \_\_mmask8 \_mm\_cmp\_round\_sd\_mask( \_\_m128d a, \_\_m128d b, int imm, int sae);

VCMPSD \_\_mmask8 \_mm\_mask\_cmp\_sd\_mask( \_\_mmask8 k1, \_\_m128d a, \_\_m128d b, int imm);

VCMPSD \_\_mmask8 \_mm\_mask\_cmp\_round\_sd\_mask( \_\_mmask8 k1, \_\_m128d a, \_\_m128d b, int imm, int sae);

(V)CMPSD \_\_m128d \_mm\_cmp\_sd( \_\_m128d a, \_\_m128d b, const int imm)

#### SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 5-3 Denormal.

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CMPSS—Compare Scalar Single-Precision Floating-Point Value

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F3 0F C2 /r ib<br>CMPSS xmm1, xmm2/m32, imm8                                   | RMI        | V/V                          | SSE                      | Compare low single-precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.  |
| VEX.NDS.128.F3.0F.WIG C2 /r ib<br>VCMPSS xmm1, xmm2, xmm3/m32,<br>imm8         | RVMI       | V/V                          | AVX                      | Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.  |
| EVEX.NDS.LIG.F3.0F.W0 C2 /r ib<br>VCMPSS k1 {k2}, xmm2,<br>xmm3/m32{sae}, imm8 | T1S        | V/V                          | AVX512F                  | Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | Imm8      |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | Imm8      |

### Description

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

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

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

The comparison predicate operand is an 8-bit immediate:

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

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

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

Note that processors with "CPUID.1H:ECX.AVX = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 5-10. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 5-10. Pseudo-Op and CMPSS Implementation**

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

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

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 5-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPS instruction. See Table 5-11, where the notations of reg1, reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPS instructions in a similar fashion by extending the syntax listed in Table 5-11.

**Table 5-11. Pseudo-Op and VCMPS Implementation**

| Pseudo-Op                           | CMPSS Implementation               |
|-------------------------------------|------------------------------------|
| VCMPEQSS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 0</i>   |
| VCMPLTSS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 1</i>   |
| VCMPLSS <i>reg1, reg2, reg3</i>     | VCMPS <i>reg1, reg2, reg3, 2</i>   |
| VCMUNORDSS <i>reg1, reg2, reg3</i>  | VCMPS <i>reg1, reg2, reg3, 3</i>   |
| VCMNEQSS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 4</i>   |
| VCMNLTSS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 5</i>   |
| VCMNLESS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 6</i>   |
| VCMORDSS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 7</i>   |
| VCMPEQ_UQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 8</i>   |
| VCMNGESS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 9</i>   |
| VCMNGTSS <i>reg1, reg2, reg3</i>    | VCMPS <i>reg1, reg2, reg3, 0AH</i> |
| VCMFALSESS <i>reg1, reg2, reg3</i>  | VCMPS <i>reg1, reg2, reg3, 0BH</i> |

**Table 5-11. Pseudo-Op and VCMPS Implementation**

| <b>Pseudo-Op</b>                    | <b>CMPS Implementation</b>         |
|-------------------------------------|------------------------------------|
| VCMPEQ_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0CH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0DH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0EH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0FH</i> |
| VCMPEQ_OSSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 10H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 11H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 12H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 13H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 14H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 15H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 16H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 17H</i> |
| VCMPEQ_OSSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 18H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 19H</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1AH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1BH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1CH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1DH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1EH</i> |
| VCMPLT_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1FH</i> |

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

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ←EQ\_OQ; OP5 ←EQ\_OQ;
- 1: OP3 ←LT\_OS; OP5 ←LT\_OS;
- 2: OP3 ←LE\_OS; OP5 ←LE\_OS;
- 3: OP3 ←UNORD\_Q; OP5 ←UNORD\_Q;
- 4: OP3 ←NEQ\_UQ; OP5 ←NEQ\_UQ;
- 5: OP3 ←NLT\_US; OP5 ←NLT\_US;
- 6: OP3 ←NLE\_US; OP5 ←NLE\_US;
- 7: OP3 ←ORD\_Q; OP5 ←ORD\_Q;
- 8: OP5 ←EQ\_UQ;
- 9: OP5 ←NGE\_US;
- 10: OP5 ←NGT\_US;
- 11: OP5 ←FALSE\_OQ;
- 12: OP5 ←NEQ\_OQ;
- 13: OP5 ←GE\_OS;
- 14: OP5 ←GT\_OS;
- 15: OP5 ←TRUE\_UQ;
- 16: OP5 ←EQ\_OS;
- 17: OP5 ←LT\_OQ;
- 18: OP5 ←LE\_OQ;

19: OP5 ← UNORD\_S;  
 20: OP5 ← NEQ\_US;  
 21: OP5 ← NLT\_UQ;  
 22: OP5 ← NLE\_UQ;  
 23: OP5 ← ORD\_S;  
 24: OP5 ← EQ\_US;  
 25: OP5 ← NGE\_UQ;  
 26: OP5 ← NGT\_UQ;  
 27: OP5 ← FALSE\_OS;  
 28: OP5 ← NEQ\_OS;  
 29: OP5 ← GE\_OQ;  
 30: OP5 ← GT\_OQ;  
 31: OP5 ← TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### VCMPS (EVEX encoded version)

CMPO ← SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or \*no writemask\*

THEN IF CMPO = TRUE

THEN DEST[0] ← 1;

ELSE DEST[0] ← 0; FI;

ELSE DEST[0] ← 0 ; zeroing-masking only

FI;

DEST[MAX\_KL-1:1] ← 0

#### CMPS (128-bit Legacy SSE version)

CMPO ← DEST[31:0] OP3 SRC[31:0];

IF CMPO = TRUE

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 00000000H; FI;

DEST[MAX\_VL-1:32] (Unmodified)

#### VCMPS (VEX.128 encoded version)

CMPO ← SRC1[31:0] OP5 SRC2[31:0];

IF CMPO = TRUE

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 00000000H; FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

VCMPS \_\_mmask8 \_\_mm\_cmp\_ss\_mask( \_\_m128 a, \_\_m128 b, int imm);

VCMPS \_\_mmask8 \_\_mm\_cmp\_round\_ss\_mask( \_\_m128 a, \_\_m128 b, int imm, int sae);

VCMPS \_\_mmask8 \_\_mm\_mask\_cmp\_ss\_mask( \_\_mmask8 k1, \_\_m128 a, \_\_m128 b, int imm);

VCMPS \_\_mmask8 \_\_mm\_mask\_cmp\_round\_ss\_mask( \_\_mmask8 k1, \_\_m128 a, \_\_m128 b, int imm, int sae);

(V)CMPS \_\_m128 \_\_mm\_cmp\_ss(\_\_m128 a, \_\_m128 b, const int imm)

#### SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 5-3, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

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

| Opcode/<br>Instruction                                 | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F 2F /r<br>COMISD xmm1, xmm2/m64                   | RM        | V/V                          | SSE2                     | Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly. |
| VEX.128.66.0F.WIG 2F /r<br>VCOMISD 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. |
| EVEX.LIG.66.0F.W1 2F /r<br>VCOMISD xmm1, xmm2/m64{sae} | T1S       | V/V                          | AVX512F                  | Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

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

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

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

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

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

### Operation

#### COMISD (all versions)

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

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
  UNORDERED: ZF,PF,CF ← 111;
```

```
  GREATER_THAN: ZF,PF,CF ← 000;
```

```
  LESS_THAN: ZF,PF,CF ← 001;
```

```
  EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VCOMISD int __mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
```

```
VCOMISD int __mm_comieq_sd (__m128d a, __m128d b)
```

```
VCOMISD int __mm_comilt_sd (__m128d a, __m128d b)
```

```
VCOMISD int __mm_comile_sd (__m128d a, __m128d b)
```

```
VCOMISD int __mm_comigt_sd (__m128d a, __m128d b)
```

```
VCOMISD int __mm_comige_sd (__m128d a, __m128d b)
```



VCOMISD int \_\_mm\_comineq\_sd (\_\_m128d a, \_\_m128d b)

#### **SIMD Floating-Point Exceptions**

Invalid (if SNaN or QNaN operands), Denormal.

#### **Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

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

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

| Opcode/<br>Instruction                              | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| OF 2F /r<br>COMISS xmm1, xmm2/m32                   | RM        | V/V                          | SSE                      | Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |
| VEX.128.OF.WIG 2F /r<br>VCOMISS 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. |
| EVEX.LIG.OF.WO 2F /r<br>VCOMISS xmm1, xmm2/m32{sae} | T1S       | V/V                          | AVX512F                  | Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Compares the single-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

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

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

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

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

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

### Operation

#### COMISS (all versions)

```

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

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

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

```

VCOMISS int \_\_mm\_comineq\_ss (\_\_m128 a, \_\_m128 b)

### **SIMD Floating-Point Exceptions**

Invalid (if SNaN or QNaN operands), Denormal.

### **Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

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

## DIVPD—Divide Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| 66 0F 5E /r<br>DIVPD xmm1, xmm2/m128   | RM         | V/V                          | SSE2                     | Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem.  |
| VEX.NDS.128.66.0F.WIG 5E /r<br>VDIVPD xmm1, xmm2, xmm3/m128                        | RVM        | V/V                          | AVX                      | Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem.  |
| VEX.NDS.256.66.0F.WIG 5E /r<br>VDIVPD ymm1, ymm2, ymm3/m256                        | RVM        | V/V                          | AVX                      | Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem.  |
| EVEX.NDS.128.66.0F.W1 5E /r<br>VDIVPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F.W1 5E /r<br>VDIVPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F.W1 5E /r<br>VDIVPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Divide packed double-precision floating-point values in zmm2 by packed double-precision FP values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.             |

### 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD divide of the double-precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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

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

VEX.128 encoded version: The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding destination are zeroed.

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or a 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAX\_VL-1:128) of the corresponding destination are unmodified.

**Operation****VDIVPD (EVEX encoded versions)**

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

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC); ; refer to Table 2-4

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

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

THEN

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE

; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VDIVPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[191:128] ← SRC1[191:128] / SRC2[191:128]

DEST[255:192] ← SRC1[255:192] / SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0;

**VDIVPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0;

**DIVPD (128-bit Legacy SSE version)**

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VDIVPD \_\_m512d \_\_mm512\_div\_pd( \_\_m512d a, \_\_m512d b);

VDIVPD \_\_m512d \_\_mm512\_mask\_div\_pd( \_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VDIVPD \_\_m512d \_\_mm512\_maskz\_div\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VDIVPD \_\_m256d \_\_mm256\_mask\_div\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VDIVPD \_\_m256d \_\_mm256\_maskz\_div\_pd( \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VDIVPD \_\_m128d \_\_mm\_mask\_div\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VDIVPD \_\_m128d \_\_mm\_maskz\_div\_pd( \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VDIVPD \_\_m512d \_\_mm512\_div\_round\_pd( \_\_m512d a, \_\_m512d b, int);  
VDIVPD \_\_m512d \_\_mm512\_mask\_div\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
VDIVPD \_\_m512d \_\_mm512\_maskz\_div\_round\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
VDIVPD \_\_m256d \_\_mm256\_div\_pd ( \_\_m256d a, \_\_m256d b);  
DIVPD \_\_m128d \_\_mm\_div\_pd ( \_\_m128d a, \_\_m128d b);

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## DIVPS—Divide Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| OF 5E /r<br>DIVPS xmm1, xmm2/m128   | RM        | V/V                          | SSE                      | Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.  |
| VEX.NDS.128.OF.WIG 5E /r<br>VDIVPS xmm1, xmm2, xmm3/m128                        | RVM       | V/V                          | AVX                      | Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.  |
| VEX.NDS.256.OF.WIG 5E /r<br>VDIVPS ymm1, ymm2, ymm3/m256                        | RVM       | V/V                          | AVX                      | Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.  |
| EVEX.NDS.128.OF.WO 5E /r<br>VDIVPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1. |
| EVEX.NDS.256.OF.WO 5E /r<br>VDIVPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1. |
| EVEX.NDS.512.OF.WO 5E /r<br>VDIVPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Divide packed single-precision floating-point values in zmm2 by packed single-precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD divide of the four, eight or sixteen packed single-precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single-precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

**Operation****VDIVPS (EVEX encoded versions)**

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

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VDIVPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[159:128] ← SRC1[159:128] / SRC2[159:128]

DEST[191:160] ← SRC1[191:160] / SRC2[191:160]

DEST[223:192] ← SRC1[223:192] / SRC2[223:192]

DEST[255:224] ← SRC1[255:224] / SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0;

**VDIVPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0



**DIVPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32] / SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64] / SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96] / SRC2[127:96]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VDIVPS \_\_m512 \_\_mm512\_div\_ps(\_\_m512 a, \_\_m512 b);  
 VDIVPS \_\_m512 \_\_mm512\_mask\_div\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VDIVPS \_\_m512 \_\_mm512\_maskz\_div\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VDIVPD \_\_m256d \_\_mm256\_mask\_div\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VDIVPD \_\_m256d \_\_mm256\_maskz\_div\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VDIVPD \_\_m128d \_\_mm\_mask\_div\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VDIVPD \_\_m128d \_\_mm\_maskz\_div\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VDIVPS \_\_m512 \_\_mm512\_div\_round\_ps(\_\_m512 a, \_\_m512 b, int);  
 VDIVPS \_\_m512 \_\_mm512\_mask\_div\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VDIVPS \_\_m512 \_\_mm512\_maskz\_div\_round\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VDIVPS \_\_m256 \_\_mm256\_div\_ps(\_\_m256 a, \_\_m256 b);  
 DIVPS \_\_m128 \_\_mm\_div\_ps(\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## DIVSD—Divide Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| F2 0F 5E /r<br>DIVSD xmm1, xmm2/m64                                       | RM         | V/V                          | SSE2                     | Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64. |
| VEX.NDS.128.F2.0F.WIG 5E /r<br>VDIVSD xmm1, xmm2, xmm3/m64                | RVM        | V/V                          | AVX                      | Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64. |
| EVEX.NDS.LIG.F2.0F.W1 5E /r<br>VDIVSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

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

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

**Operation****VDIVSD (EVEX encoded version)**

```

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

```

**VDIVSD (VEX.128 encoded version)**

```

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

```

**DIVSD (128-bit Legacy SSE version)**

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd(__m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd(__m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## DIVSS—Divide Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| F3 0F 5E /r<br>DIVSS xmm1, xmm2/m32                                       | RM         | V/V                          | SSE                      | Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32. |
| VEX.NDS.128.F3.0F.WIG 5E /r<br>VDIVSS xmm1, xmm2, xmm3/m32                | RVM        | V/V                          | AVX                      | Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32. |
| EVEX.NDS.LIG.F3.0F.W0 5E /r<br>VDIVSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

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

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

**Operation****VDIVSS (EVEX encoded version)**

```

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

```

**VDIVSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**DIVSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] / SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VDIVSS __m128 __mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 __mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 __mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 __mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 __mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 __mm_div_ss(__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 8A /r<br>VCOMPRESSPD xmm1/m128 {k1}{z},<br>xmm2 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.    |
| EVEX.256.66.0F38.W1 8A /r<br>VCOMPRESSPD ymm1/m256 {k1}{z},<br>ymm2 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.    |
| EVEX.512.66.0F38.W1 8A /r<br>VCOMPRESSPD zmm1/m512 {k1}{z},<br>zmm2 | T1S       | V/V                          | AVX512F                  | Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VCOMPRESSPD (EVEX encoded versions) store form**

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

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

  THEN

    DEST[k+SIZE-1:k] ← SRC[i+63:i]

    k ← k + SIZE

  FI;

ENDFOR

**VCOMPRESSPD (EVEX encoded versions) reg-reg form**

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

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+63:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPD \_\_m512d \_\_mm512\_mask\_compress\_pd( \_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD \_\_m512d \_\_mm512\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD void \_\_mm512\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD \_\_m256d \_\_mm256\_mask\_compress\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD \_\_m256d \_\_mm256\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD void \_\_mm256\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD \_\_m128d \_\_mm\_mask\_compress\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VCOMPRESSPD \_\_m128d \_\_mm\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m128d a);

VCOMPRESSPD void \_\_mm\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

#UD                   If EVEX.vvvv != 1111B.

**VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory**

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 8A /r<br>VCOMPRESSPS xmm1/m128 {k1}{z},<br>xmm2 | T1S        | V/V                          | AVX512VL<br>AVX512F      | Compress packed single-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.    |
| EVEX.256.66.0F38.W0 8A /r<br>VCOMPRESSPS ymm1/m256 {k1}{z},<br>ymm2 | T1S        | V/V                          | AVX512VL<br>AVX512F      | Compress packed single-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.    |
| EVEX.512.66.0F38.W0 8A /r<br>VCOMPRESSPS zmm1/m512 {k1}{z},<br>zmm2 | T1S        | V/V                          | AVX512F                  | Compress packed single-precision floating-point values from zmm2 using control mask k1 to zmm1/m512. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

**Description**

Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VCOMPRESSPS (EVEX encoded versions) store form**

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

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+31:i]

      k ← k + SIZE

  FI;

ENDFOR;



**VCOMPRESSPS (EVEX encoded versions) reg-reg form**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k] ← SRC[i+31:i]
            k ← k + SIZE
        FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] ← 0
FI
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCOMPRESSPS __m512 __mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 __mm512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void __mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 __mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 __mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void __mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 __mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 __mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void __mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

#UD If EVEX.vvvv != 1111B.

## CVTDDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| F3 0F E6 /r<br>CVTDDQ2PD xmm1, xmm2/m64                                  | RM        | V/V                          | SSE2                     | Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.                                |
| VEX.128.F3.0F.WIG E6 /r<br>VCVTDDQ2PD xmm1, xmm2/m64                     | RM        | V/V                          | AVX                      | Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.                                |
| VEX.256.F3.0F.WIG E6 /r<br>VCVTDDQ2PD ymm1, xmm2/m128                    | RM        | V/V                          | AVX                      | Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.                              |
| EVEX.128.F3.0F.W0 E6 /r<br>VCVTDDQ2PD xmm1 {k1}{z},<br>xmm2/m128/m32bcst | HV        | V/V                          | AVX512VL<br>AVX512F      | Convert 2 packed signed doubleword integers from xmm2/m128/m32bcst to eight packed double-precision floating-point values in xmm1 with writemask k1.     |
| EVEX.256.F3.0F.W0 E6 /r<br>VCVTDDQ2PD ymm1 {k1}{z},<br>xmm2/m128/m32bcst | HV        | V/V                          | AVX512VL<br>AVX512F      | Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double-precision floating-point values in ymm1 with writemask k1.         |
| EVEX.512.F3.0F.W0 E6 /r<br>VCVTDDQ2PD zmm1 {k1}{z},<br>ymm2/m256/m32bcst | HV        | V/V                          | AVX512F                  | Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double-precision floating-point values in the destination operand (the first operand).

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

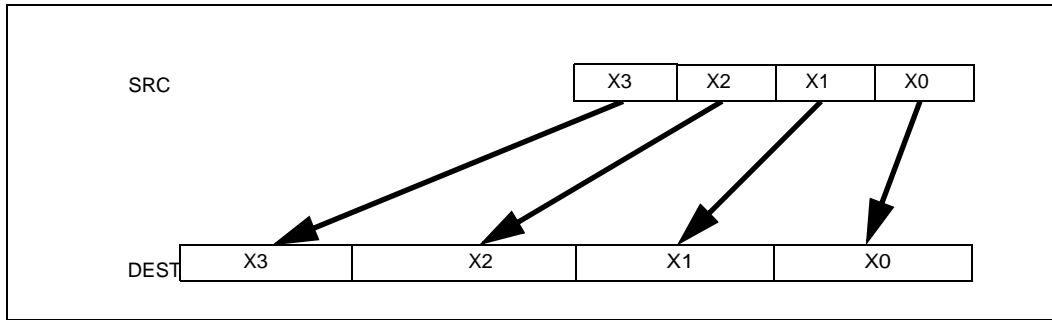


Figure 5-11. CVTDDQ2PD (VEX.256 encoded version)

### Operation

#### VCVTDQ2PD (EVEX encoded versions) when src operand is a register

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

    ELSE

      IF \*merging-masking\* ; merging-masking

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

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VCVTDQ2PD (EVEX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1)

        THEN

          DEST[i+63:i] ←

          Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])

        ELSE

          DEST[i+63:i] ←

          Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

        FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

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

      ELSE ; zeroing-masking

```

                DEST[i+63:i] ← 0
            FI
        ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**VCVTDQ2PD (VEX.256 encoded version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAX_VL-1:256] ← 0

```

**VCVTDQ2PD (VEX.128 encoded version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] ← 0

```

**CVTDQ2PD (128-bit Legacy SSE version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTDQ2PD __m512d __mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d __mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d __mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d __mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d __mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m256d __mm256_cvtepi32_pd( __m128i src)
CVTDQ2PD __m128d __mm_cvtepi32_pd( __m128i src)

```

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 5;

EVEX-encoded instructions, see Exceptions Type E5.

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

## CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

| Opcode<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| OF 5B /r<br>CVTDQ2PS xmm1, xmm2/m128                                     | RM        | V/V                          | SSE2                     | Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.                                  |
| VEX.128.OF.WIG 5B /r<br>VCVTDQ2PS xmm1, xmm2/m128                        | RM        | V/V                          | AVX                      | Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.                                  |
| VEX.256.OF.WIG 5B /r<br>VCVTDQ2PS ymm1, ymm2/m256                        | RM        | V/V                          | AVX                      | Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.                                |
| EVEX.128.OF.W0 5B /r<br>VCVTDQ2PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single-precision floating-point values in xmm1 with writemask k1.       |
| EVEX.256.OF.W0 5B /r<br>VCVTDQ2PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single-precision floating-point values in ymm1 with writemask k1.     |
| EVEX.512.OF.W0 5B /r<br>VCVTDQ2PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single-precision floating-point values in the destination operand.

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

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

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

**Operation****VCVTDQ2PS (EVEX encoded versions) when SRC operand is a register**

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

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

THEN

SET\_RM(EVEX.RC); ; refer to Table 2-4

ELSE

SET\_RM(MXCSR.RM); ; refer to Table 2-4

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTDQ2PS (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[159:128] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[159:128])  
 DEST[191:160] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[191:160])  
 DEST[223:192] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[223:192])  
 DEST[255:224] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[255:224])  
 DEST[MAX\_VL-1:256] ← 0

**VCVTDQ2PS (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[MAX\_VL-1:128] ← 0

**CVTDQ2PS (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTDQ2PS \_\_m512 \_\_mm512\_cvtepi32\_ps( \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_mask\_cvtepi32\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_maskz\_cvtepi32\_ps( \_\_mmask16 k, \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_cvt\_roundepsi32\_ps( \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_mask\_cvt\_roundepsi32\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_maskz\_cvt\_roundepsi32\_ps( \_\_mmask16 k, \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m256 \_\_mm256\_mask\_cvtepi32\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256i a);  
 VCVTDQ2PS \_\_m256 \_\_mm256\_maskz\_cvtepi32\_ps( \_\_mmask8 k, \_\_m256i a);  
 VCVTDQ2PS \_\_m128 \_\_mm\_mask\_cvtepi32\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128i a);  
 VCVTDQ2PS \_\_m128 \_\_mm\_maskz\_cvtepi32\_ps( \_\_mmask8 k, \_\_m128i a);  
 CVTDQ2PS \_\_m256 \_\_mm256\_cvtepi32\_ps( \_\_m256i src)  
 CVTDQ2PS \_\_m128 \_\_mm\_cvtepi32\_ps( \_\_m128i src)

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

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

## CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| F2 0F E6 /r<br>CVTPD2DQ xmm1, xmm2/m128                                     | RM         | V/V                          | SSE2                     | Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.                                      |
| VEX.128.F2.0F.WIG E6 /r<br>VCVTPD2DQ xmm1, xmm2/m128                        | RM         | V/V                          | AVX                      | Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.                                      |
| VEX.256.F2.0F.WIG E6 /r<br>VCVTPD2DQ xmm1, ymm2/m256                        | RM         | V/V                          | AVX                      | Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.                                    |
| EVEX.128.F2.0F.W1 E6 /r<br>VCVTPD2DQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.     |
| EVEX.256.F2.0F.W1 E6 /r<br>VCVTPD2DQ xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.   |
| EVEX.512.F2.0F.W1 E6 /r<br>VCVTPD2DQ ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.



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

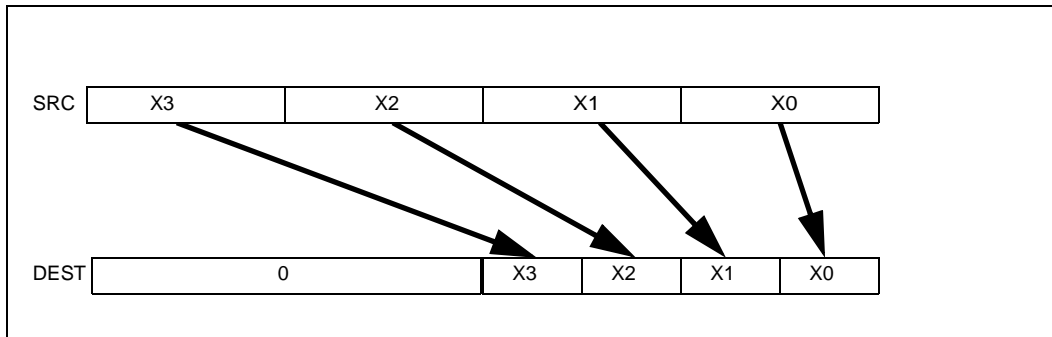


Figure 5-12. VCVTPD2DQ (VEX.256 encoded version)

### Operation

**VCVTPD2DQ (EVEX encoded versions) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0

**VCVTPD2DQ (EVEX encoded versions) when src operand is a memory source**

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

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

ELSE

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[k+63:k])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0

**VCVTPD2DQ (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

DEST[63:32] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[127:64])

DEST[95:64] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[191:128])

DEST[127:96] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[255:192])

DEST[MAX\_VL-1:128] ← 0

**VCVTPD2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

DEST[63:32] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[127:64])

DEST[MAX\_VL-1:64] ← 0

**CVTPD2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

DEST[63:32] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[127:64])

DEST[127:64] ← 0

DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPD2DQ \_\_m256i \_\_mm512\_cvtpd\_epi32( \_\_m512d a);

VCVTPD2DQ \_\_m256i \_\_mm512\_mask\_cvtpd\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m512d a);

VCVTPD2DQ \_\_m256i \_\_mm512\_maskz\_cvtpd\_epi32( \_\_mmask8 k, \_\_m512d a);

VCVTPD2DQ \_\_m256i \_\_mm512\_cvt\_roundpd\_epi32( \_\_m512d a, int r);

VCVTPD2DQ \_\_m256i \_\_mm512\_mask\_cvt\_roundpd\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m512d a, int r);

VCVTPD2DQ \_\_m256i \_\_mm512\_maskz\_cvt\_roundpd\_epi32( \_\_mmask8 k, \_\_m512d a, int r);

VCVTPD2DQ \_\_m128i \_\_mm256\_mask\_cvtpd\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m256d a);

VCVTPD2DQ \_\_m128i \_\_mm256\_maskz\_cvtpd\_epi32( \_\_mmask8 k, \_\_m256d a);

VCVTPD2DQ \_\_m128i \_\_mm\_mask\_cvtpd\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128d a);

VCVTPD2DQ \_\_m128i \_\_mm\_maskz\_cvtpd\_epi32( \_\_mmask8 k, \_\_m128d a);

VCVTPD2DQ \_\_m128i \_\_mm256\_cvtpd\_epi32 (\_\_m256d src)  
CVTPD2DQ \_\_m128i \_\_mm\_cvtpd\_epi32 (\_\_m128d src)

### **SIMD Floating-Point Exceptions**

Invalid, Precision

### **Other Exceptions**

See Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

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

## CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| 66 0F 5A /r<br>CVTPD2PS xmm1, xmm2/m128                                     | RM        | V/V                          | SSE2                     | Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.                                |
| VEX.128.66.0F.WIG 5A /r<br>VCVTPD2PS xmm1, xmm2/m128                        | RM        | V/V                          | AVX                      | Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.                                |
| VEX.256.66.0F.WIG 5A /r<br>VCVTPD2PS xmm1, ymm2/m256                        | RM        | V/V                          | AVX                      | Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1.                              |
| EVEX.128.66.0F.W1 5A /r<br>VCVTPD2PS xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two single-precision floating-point values in xmm1 with writemask k1.     |
| EVEX.256.66.0F.W1 5A /r<br>VCVTPD2PS xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four single-precision floating-point values in xmm1 with writemask k1.   |
| EVEX.512.66.0F.W1 5A /r<br>VCVTPD2PS ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight single-precision floating-point values in ymm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed single-precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

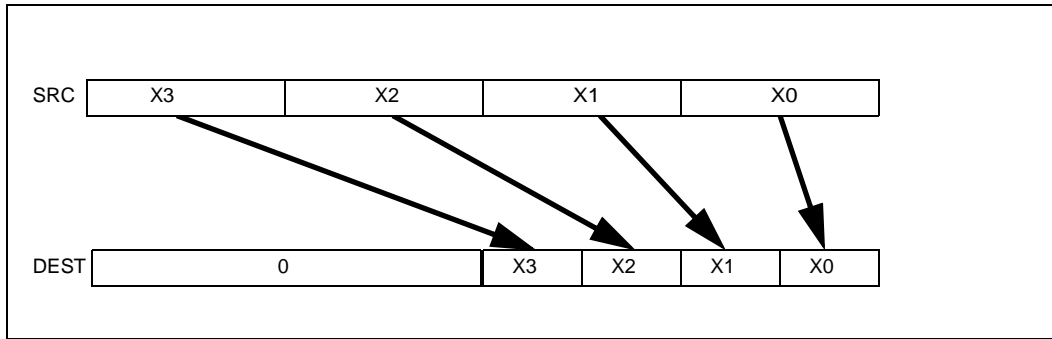


Figure 5-13. VCVTPD2PS (VEX.256 encoded version)

### Operation

#### VCVTPD2PS (EVEX encoded version) when src operand is a register

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[i+31:i] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0

**VCVTPD2PS (EVEX encoded version) when src operand is a memory source**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTPD2PS (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
DEST[MAX_VL-1:128] ← 0

```

**VCVTPD2PS (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAX_VL-1:64] ← 0

```

**CVTPD2PS (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] ← 0
DEST[MAX_VL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2PS __m256 __mm512_cvtpd_ps( __m512d a);
VCVTPD2PS __m256 __mm512_mask_cvtpd_ps( __m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_maskz_cvtpd_ps( __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_cvt_roundpd_ps( __m512d a, int r);
VCVTPD2PS __m256 __mm512_mask_cvt_roundpd_ps( __m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 __mm512_maskz_cvt_roundpd_ps( __mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 __mm256_mask_cvtpd_ps( __m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm256_maskz_cvtpd_ps( __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm_mask_cvtpd_ps( __m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm_maskz_cvtpd_ps( __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm256_cvtpd_ps( __m256d a)

```

CVTPD2PS \_\_m128 \_mm\_cvtpd\_ps (\_\_m128d a)

### **SIMD Floating-Point Exceptions**

Invalid, Precision, Underflow, Overflow, Denormal

### **Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

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

## VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W1 7B /r<br>VCVTPD2QQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.     |
| EVEX.256.66.0F.W1 7B /r<br>VCVTPD2QQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.   |
| EVEX.512.66.0F.W1 7B /r<br>VCVTPD2QQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV        | V/V                          | AVX512DQ                 | Convert eight packed double-precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

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

### Operation

#### VCVTPD2QQ (EVEX encoded version) when src operand is a register

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking



```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### VCVTPD2QQ (EVEX encoded version) when src operand is a memory source

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2QQ __m512i __mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i __mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i __mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i __mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i __mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i __mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m128i __mm_mask_cvtpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i __mm_maskz_cvtpd_epi64( __mmask8 k, __m128d a);
VCVTPD2QQ __m256i __mm256_cvtpd_epi64( __m256d src)
VCVTPD2QQ __m128i __mm_cvtpd_epi64( __m128d src)

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

| Opcode<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.OF.W1 79 /r<br>VCVTPD2UDQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.     |
| EVEX.256.OF.W1 79 /r<br>VCVTPD2UDQ xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.   |
| EVEX.512.OF.W1 79 /r<br>VCVTPD2UDQ ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

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

### Operation

#### VCVTPD2UDQ (EVEX encoded versions) when src2 operand is a register

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 32$

$k \leftarrow j * 64$

IF k1[j] OR \*no writemask\*

THEN

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[k+63:k])

ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

#### VCVTPD2UDQ (EVEX encoded versions) when src operand is a memory source

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

```

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                    Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])
                ELSE
                    DEST[i+31:i] ←
                    Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2UDQ __m256i __mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i __mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i __mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i __mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i __mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i __mm512_maskz_cvt_roundpd_epu32( __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i __mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i __mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W1 79 /r<br>VCVTPD2UQQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.     |
| EVEX.256.66.0F.W1 79 /r<br>VCVTPD2UQQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1. |
| EVEX.512.66.0F.W1 79 /r<br>VCVTPD2UQQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV         | V/V                          | AVX512DQ                 | Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTPD2UQQ (EVEX encoded versions) when src operand is a register

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+63:i] \leftarrow$

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

VCVTPD2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
                    Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
                    Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                                  ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2UQQ __m512i __mm512_cvtpd_epu64( __m512d a);
VCVTPD2UQQ __m512i __mm512_mask_cvtpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_maskz_cvtpd_epu64( __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_cvt_roundpd_epu64( __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_mask_cvt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_maskz_cvt_roundpd_epu64( __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i __mm256_mask_cvtpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i __mm256_maskz_cvtpd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m128i __mm_mask_cvtpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i __mm_maskz_cvtpd_epu64( __mmask8 k, __m128d a);
VCVTPD2UQQ __m256i __mm256_cvtpd_epu64( __m256d src)
VCVTPD2UQQ __m128i __mm_cvtpd_epu64( __m128d src)

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 1313 /r<br>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.      |
| VEX.256.66.0F38.W0 1313 /r<br>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.    |
| EVEX.128.66.0F38.W0 1313 /r<br>VCVTPH2PS xmm1 {k1}{z}, xmm2/m64           | HVM        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.     |
| EVEX.256.66.0F38.W0 1313 /r<br>VCVTPH2PS ymm1 {k1}{z},<br>xmm2/m128       | HVM        | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.   |
| EVEX.512.66.0F38.W0 1313 /r<br>VCVTPH2PS zmm1 {k1}{z},<br>ymm2/m256 {sae} | HVM        | V/V                          | AVX512F                  | Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1. |

### Instruction Operand Encoding

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

### Description

Converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single-precision floating-point values and writes the converted values into the destination operand (the first operand).

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.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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 and EVEX.vvvv are reserved (must be 1111b).

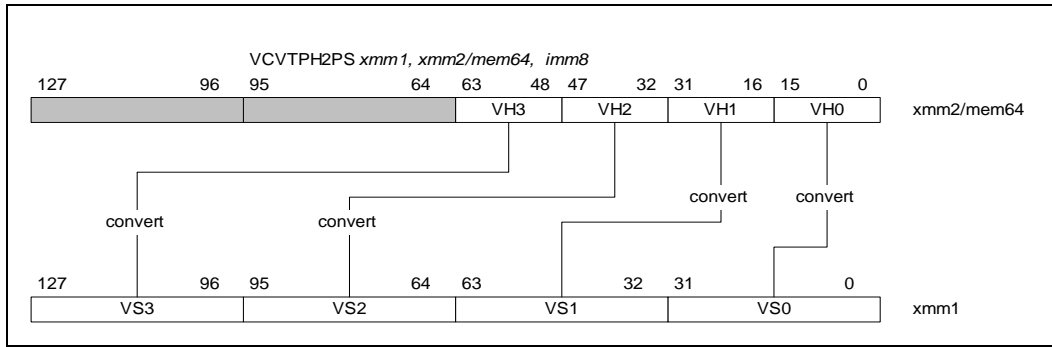


Figure 5-14. VCVTPH2PS (128-bit Version)

**Operation**

```

vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}

```

**VCVTPH2PS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  k ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ←

      vCvt\_h2s(SRC[k+15:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 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]);

DEST[MAX\_VL-1:256] ← 0

**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[MAX\_VL-1:128] ← 0

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPH2PS \_\_m512 \_\_mm512\_cvtph\_ps( \_\_m256i a);  
 VCVTPH2PS \_\_m512 \_\_mm512\_mask\_cvtph\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m256i a);  
 VCVTPH2PS \_\_m512 \_\_mm512\_maskz\_cvtph\_ps(\_\_mmask16 k, \_\_m256i a);  
 VCVTPH2PS \_\_m512 \_\_mm512\_cvt\_roundph\_ps( \_\_m256i a, int sae);  
 VCVTPH2PS \_\_m512 \_\_mm512\_mask\_cvt\_roundph\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m256i a, int sae);  
 VCVTPH2PS \_\_m512 \_\_mm512\_maskz\_cvt\_roundph\_ps( \_\_mmask16 k, \_\_m256i a, int sae);  
 VCVTPH2PS \_\_m256 \_\_mm256\_mask\_cvtph\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m128i a);  
 VCVTPH2PS \_\_m256 \_\_mm256\_maskz\_cvtph\_ps(\_\_mmask8 k, \_\_m128i a);  
 VCVTPH2PS \_\_m128 \_\_mm\_mask\_cvtph\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128i a);  
 VCVTPH2PS \_\_m128 \_\_mm\_maskz\_cvtph\_ps(\_\_mmask8 k, \_\_m128i a);  
 VCVTPH2PS \_\_m128 \_\_mm\_cvtph\_ps( \_\_m128i m1);  
 VCVTPH2PS \_\_m256 \_\_mm256\_cvtph\_ps( \_\_m128i m1)

**SIMD Floating-Point Exceptions**

Invalid

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD If VEX.W=1.

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



## VCVTSP2PH—Convert Single-Precision FP value to 16-bit FP value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|-----------------------------|--------------------------|--|
| VEX.128.66.0F3A.W0 1D 1D/r ib<br>VCVTSP2PH xmm1/m64, xmm2,<br>imm8                | MRI        | V/V                         | F16C                     | Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.     |
| VEX.256.66.0F3A.W0 1D1D /r ib<br>VCVTSP2PH xmm1/m128, ymm2,<br>imm8               | MRI        | V/V                         | F16C                     | Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.   |
| EVEX.128.66.0F3A.W0 1D1D /r ib<br>VCVTSP2PH xmm1/m64 {k1}{z},<br>xmm2, imm8       | HVM        | V/V                         | AVX512VL<br>AVX512F      | Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.     |
| EVEX.256.66.0F3A.W0 1D1D /r ib<br>VCVTSP2PH xmm1/m128 {k1}{z},<br>ymm2, imm8      | HVM        | V/V                         | AVX512VL<br>AVX512F      | Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.   |
| EVEX.512.66.0F3A.W0 1D1D /r ib<br>VCVTSP2PH ymm1/m256 {k1}{z},<br>zmm2{sae}, imm8 | HVM        | V/V                         | AVX512F                  | Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| MRI   | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |
| HVM   | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |

### Description

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e. tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

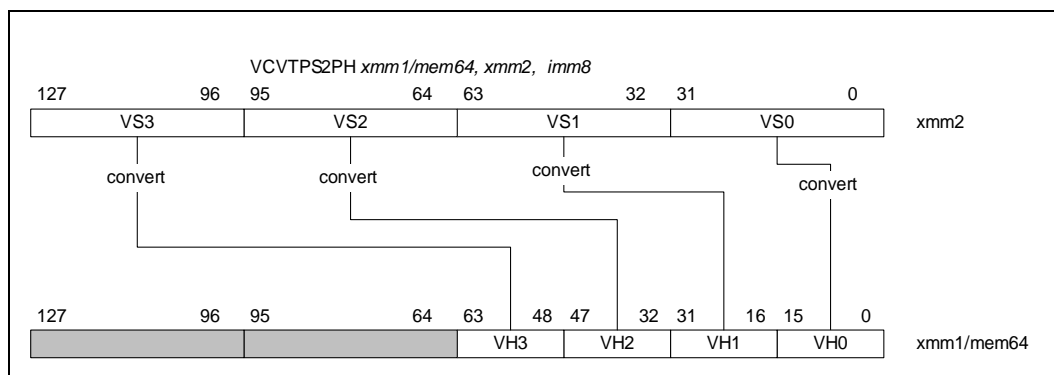


Figure 5-15. VCVTSP2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-12.

**Table 5-12. 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      |                 |

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAX\_VL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAX\_VL-1:256/128/64) of the corresponding destination register are zeroed.

### Operation

```
vCvt_s2h(SRC1[31:0])
{
  IF Imm[2] = 0
  THEN   ; using Imm[1:0] for rounding control, see Table 5-12
    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 (EVEX encoded versions) when dest is a register

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

```
FOR j ← 0 TO KL-1
  i ← j * 16
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ←
      vCvt_s2h(SRC[k+31:k])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+15:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
```

**VCVTPS2PH (EVEX encoded versions) when dest is memory**

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

```

FOR j ← 0 TO KL-1
  i ← j * 16
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ←
      vCvt_s2h(SRC[k+31:k])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**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[MAX_VL-1:128] ← 0

```

**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[MAX_VL-1:64] ← 0

```

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPS2PH __m256i __mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i __mm512_mask_cvtps_ph(__m256i s, __mmask16 k, __m512 a);
VCVTPS2PH __m256i __mm512_maskz_cvtps_ph(__mmask16 k, __m512 a);
VCVTPS2PH __m256i __mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i __mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m256i __mm512_maskz_cvt_roundps_ph(__mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m128i __mm256_mask_cvtps_ph(__m128i s, __mmask8 k, __m256 a);
VCVTPS2PH __m128i __mm256_maskz_cvtps_ph(__mmask8 k, __m256 a);
VCVTPS2PH __m128i __mm_mask_cvtps_ph(__m128i s, __mmask8 k, __m128 a);
VCVTPS2PH __m128i __mm_maskz_cvtps_ph(__mmask8 k, __m128 a);
VCVTPS2PH __m128i __mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i __mm256_cvtps_ph(__m256 m1, const int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD If VEX.W=1.

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

## CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F 5B /r<br>CVTPS2DQ xmm1, xmm2/m128                                     | RM         | V/V                          | SSE2                     | Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.  |
| VEX.128.66.0F.WIG 5B /r<br>VCVTPS2DQ xmm1, xmm2/m128                        | RM         | V/V                          | AVX                      | Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.  |
| VEX.256.66.0F.WIG 5B /r<br>VCVTPS2DQ ymm1, ymm2/m256                        | RM         | V/V                          | AVX                      | Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.                                      |
| EVEX.128.66.0F.W0 5B /r<br>VCVTPS2DQ xmm1 {k1}{z},<br>xmm2/m128/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.       |
| EVEX.256.66.0F.W0 5B /r<br>VCVTPS2DQ ymm1 {k1}{z},<br>ymm2/m256/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.     |
| EVEX.512.66.0F.W0 5B /r<br>VCVTPS2DQ zmm1 {k1}{z},<br>zmm2/m512/m32bcst{er} | FV         | V/V                          | AVX512F                  | Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

### Operation

#### VCVTPS2DQ (encoded versions) when src operand is a register

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VCVTPS2DQ (EVEX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO 15

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTPS2DQ (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[159:128] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[159:128])  
 DEST[191:160] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[191:160])  
 DEST[223:192] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[223:192])  
 DEST[255:224] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[255:224])

**VCVTPS2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[MAX\_VL-1:128] ← 0

**CVTPS2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2DQ \_\_m512i \_\_mm512\_cvtps\_epi32( \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_mask\_cvtps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvtps\_epi32( \_\_mmask16 k, \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_cvt\_roundps\_epi32( \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_mask\_cvt\_roundps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvt\_roundps\_epi32( \_\_mmask16 k, \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_mask\_cvtps\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256 a);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_maskz\_cvtps\_epi32( \_\_mmask8 k, \_\_m256 a);  
 VCVTPS2DQ \_\_m128i \_\_mm\_mask\_cvtps\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128 a);  
 VCVTPS2DQ \_\_m128i \_\_mm\_maskz\_cvtps\_epi32( \_\_mmask8 k, \_\_m128 a);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_cvtps\_epi32( \_\_m256 a)  
 CVTPS2DQ \_\_m128i \_\_mm\_cvtps\_epi32( \_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

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

## VCVTPS2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.OF.W0 79 /r<br>VCVTPS2UDQ xmm1 {k1}{z},<br>xmm2/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.       |
| EVEX.256.OF.W0 79 /r<br>VCVTPS2UDQ ymm1 {k1}{z},<br>ymm2/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.     |
| EVEX.512.OF.W0 79 /r<br>VCVTPS2UDQ zmm1 {k1}{z},<br>zmm2/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts sixteen packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask  $k1$ .

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

### Operation

**VCVTPS2UDQ (EVEX encoded versions) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+31:i] \leftarrow$

Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

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



```

        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTSP2UDQ (EVEX encoded versions) when src operand is a memory source**  
(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                    Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
                    Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSP2UDQ __m512i __mm512_cvtps_epu32( __m512 a);
VCVTSP2UDQ __m512i __mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTSP2UDQ __m512i __mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTSP2UDQ __m512i __mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTSP2UDQ __m512i __mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTSP2UDQ __m512i __mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTSP2UDQ __m256i __mm256_cvtps_epu32( __m256d a);
VCVTSP2UDQ __m256i __mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTSP2UDQ __m256i __mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTSP2UDQ __m128i __mm_cvtps_epu32( __m128 a);
VCVTSP2UDQ __m128i __mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTSP2UDQ __m128i __mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTQPS2Q—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEH.128.66.0F.W0 7B /r<br>VCVTQPS2Q xmm1 {k1}{z},<br>xmm2/m64/m32bcst      | HV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.      |
| EVEH.256.66.0F.W0 7B /r<br>VCVTQPS2Q ymm1 {k1}{z},<br>xmm2/m128/m32bcst     | HV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.   |
| EVEH.512.66.0F.W0 7B /r<br>VCVTQPS2Q zmm1 {k1}{z},<br>ymm2/m256/m32bcst{er} | HV         | V/V                          | AVX512DQ                 | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts eight packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTQPS2Q (EVEH encoded versions) when src operand is a register**

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

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

THEN

SET\_RM(EVEH.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_QuadInteger(SRC[k+31:k])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### VCVTSP2QQ (EVEX encoded versions) when src operand is a memory source

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
                    Convert_Single_Precision_To_QuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
                    Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSP2QQ __m512i __mm512_cvtps_epi64( __m512 a);
VCVTSP2QQ __m512i __mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTSP2QQ __m512i __mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTSP2QQ __m512i __mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTSP2QQ __m512i __mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTSP2QQ __m512i __mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTSP2QQ __m256i __mm256_cvtps_epi64( __m256 a);
VCVTSP2QQ __m256i __mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTSP2QQ __m256i __mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTSP2QQ __m128i __mm_cvtps_epi64( __m128 a);
VCVTSP2QQ __m128i __mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTSP2QQ __m128i __mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3

#UD If EVEX.vvvv != 1111B.

## VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F.W0 79 /r<br>VCVTPS2UQQ xmm1 {k1}{z},<br>xmm2/m64/m32bcst      | HV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.      |
| EVEX.256.66.0F.W0 79 /r<br>VCVTPS2UQQ ymm1 {k1}{z},<br>xmm2/m128/m32bcst     | HV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.   |
| EVEX.512.66.0F.W0 79 /r<br>VCVTPS2UQQ zmm1 {k1}{z},<br>ymm2/m256/m32bcst{er} | HV        | V/V                          | AVX512DQ                 | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTPS2UQQ (EVEX encoded versions) when src operand is a register

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

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

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking

```

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### VCVT<sub>PS2UQ</sub>Q (EVEX encoded versions) when src operand is a memory source

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
                    Convert_Single_Precision_To_UQuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
                    Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPS2UQQ __m512i __mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i __mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i __mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i __mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i __mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i __mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i __mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i __mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3

#UD If EVEX.vvvv != 1111B.

## CVTTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| OF 5A /r<br>CVTTPS2PD xmm1, xmm2/m64                                       | RM        | V/V                          | SSE2                     | Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.                                |
| VEX.128.OF.WIG 5A /r<br>VCVTTPS2PD xmm1, xmm2/m64                          | RM        | V/V                          | AVX                      | Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.                                |
| VEX.256.OF.WIG 5A /r<br>VCVTTPS2PD ymm1, xmm2/m128                         | RM        | V/V                          | AVX                      | Convert four packed single-precision floating-point values in xmm2/m128 to four packed double-precision floating-point values in ymm1.                             |
| EVEX.128.OF.W0 5A /r<br>VCVTTPS2PD xmm1 {k1}{z},<br>xmm2/m64/m32bcst       | HV        | V/V                          | AVX512VL<br>AVX512F      | Convert two packed single-precision floating-point values in xmm2/m64/m32bcst to packed double-precision floating-point values in xmm1 with writemask k1.          |
| EVEX.256.OF.W0 5A /r<br>VCVTTPS2PD ymm1 {k1}{z},<br>xmm2/m128/m32bcst      | HV        | V/V                          | AVX512VL                 | Convert four packed single-precision floating-point values in xmm2/m128/m32bcst to packed double-precision floating-point values in ymm1 with writemask k1.        |
| EVEX.512.OF.W0 5A /r<br>VCVTTPS2PD zmm1 {k1}{z},<br>ymm2/m256/m32bcst{sae} | HV        | V/V                          | AVX512F                  | Convert eight packed single-precision floating-point values in ymm2/m256/b32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts two, four or eight packed single-precision floating-point values in the source operand (second operand) to two, four or eight packed double-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

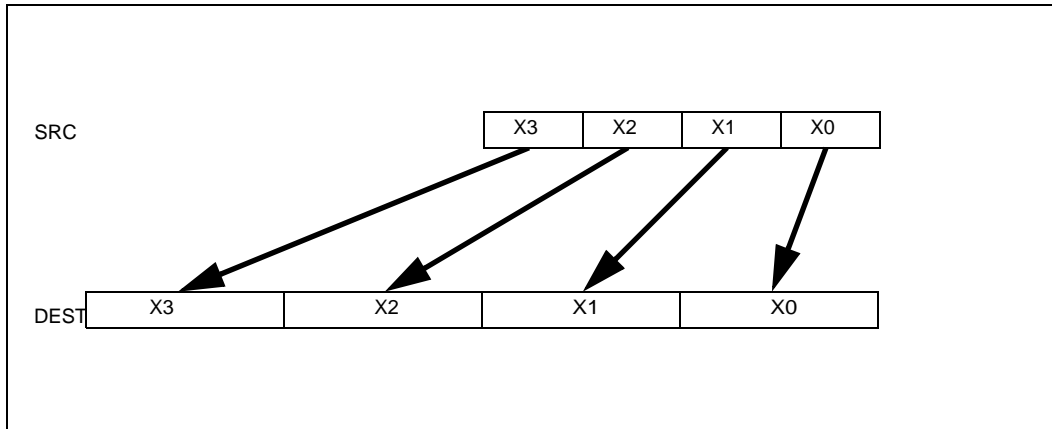


Figure 5-16. CVTSP2PD (VEX.256 encoded version)

**Operation****VCVTSP2PD (EVEX encoded versions) when src operand is a register**

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

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTSP2PD (EVEX encoded versions) when src operand is a memory source**

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

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])

ELSE

DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

FI;

ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+63:i] remains unchanged*
            ELSE                         ; zeroing-masking
                DEST[j+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTSP2PD (VEX.256 encoded version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAX_VL-1:256] ← 0

```

**VCVTSP2PD (VEX.128 encoded version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] ← 0

```

**CVTSP2PD (128-bit Legacy SSE version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTSP2PD __m512d __mm512_cvtps_pd( __m256 a);
VCVTSP2PD __m512d __mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTSP2PD __m512d __mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTSP2PD __m512d __mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTSP2PD __m512d __mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTSP2PD __m512d __mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTSP2PD __m256d __mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTSP2PD __m256d __mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTSP2PD __m128d __mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTSP2PD __m128d __mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTSP2PD __m256d __mm256_cvtps_pd( __m128 a)
CVTSP2PD __m128d __mm_cvtps_pd( __m128 a)

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3.

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



## VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F.W1 E6 /r<br>VCVTQQ2PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.         |
| EVEX.256.F3.0F.W1 E6 /r<br>VCVTQQ2PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.        |
| EVEX.512.F3.0F.W1 E6 /r<br>VCVTQQ2PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV         | V/V                          | AVX512DQ                 | Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source**

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b == 1)

        THEN

          DEST[i+63:i] ←

          Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[63:0])

        ELSE

          DEST[i+63:i] ←

          Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

        FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

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

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTQQ2PD \_\_m512d \_\_mm512\_cvtepi64\_pd( \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_mask\_cvtepi64\_pd( \_\_m512d s, \_\_mmask16 k, \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_maskz\_cvtepi64\_pd( \_\_mmask16 k, \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_cvt\_roundepi64\_pd( \_\_m512i a, int r);

VCVTQQ2PD \_\_m512d \_\_mm512\_mask\_cvt\_roundepi\_ps( \_\_m512d s, \_\_mmask8 k, \_\_m512i a, int r);

VCVTQQ2PD \_\_m512d \_\_mm512\_maskz\_cvt\_roundepi64\_pd( \_\_mmask8 k, \_\_m512i a, int r);

VCVTQQ2PD \_\_m256d \_\_mm256\_mask\_cvtepi64\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256i a);

VCVTQQ2PD \_\_m256d \_\_mm256\_maskz\_cvtepi64\_pd( \_\_mmask8 k, \_\_m256i a);

VCVTQQ2PD \_\_m128d \_\_mm\_mask\_cvtepi64\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128i a);

VCVTQQ2PD \_\_m128d \_\_mm\_maskz\_cvtepi64\_pd( \_\_mmask8 k, \_\_m128i a);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.0F.W1 5B /r<br>VCVTQQ2PS xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed quadword integers from xmm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.         |
| EVEX.256.0F.W1 5B /r<br>VCVTQQ2PS xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed quadword integers from ymm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.        |
| EVEX.512.0F.W1 5B /r<br>VCVTQQ2PS ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV        | V/V                          | AVX512DQ                 | Convert eight packed quadword integers from zmm2/mem to eight packed single-precision floating-point values in ymm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1      | Operand 2     | Operand 3 | Operand 4 |
|-------|----------------|---------------|-----------|-----------|
| FV    | ModRM:rreg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts packed quadword integers in the source operand (second operand) to packed single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

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

### Operation

**VCVTQQ2PS (EVEX encoded versions) when src operand is a register**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[k+31:k] ←
      Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[k+31:k] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[k+31:k] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTQQ2PS (EVEX encoded versions) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[k+31:k] ←
          Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[k+31:k] ←
          Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[j+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[k+31:k] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[k+31:k] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL/2] ← 0
  
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTQQ2PS __m256 __mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 __mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 __mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 __mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 __mm512_mask_cvt_roundepi64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 __mm512_maskz_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 __mm256_cvtepi64_ps( __m256i a);
VCVTQQ2PS __m128 __mm256_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 __mm256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQQ2PS __m128 __mm_cvtepi64_ps( __m128i a);
VCVTQQ2PS __m128 __mm_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 __mm_maskz_cvtepi64_ps( __mmask8 k, __m128i a);
  
```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

| Opcode/<br>Instruction                                 | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| F2 0F 2D /r<br>CVTSD2SI r32, xmm1/m64                  | RM        | V/V                          | SSE2                     | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.                  |
| F2 REX.W 0F 2D /r<br>CVTSD2SI r64, xmm1/m64            | RM        | V/N.E.                       | SSE2                     | Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64. |
| VEX.128.F2.0F.W0 2D /r<br>VCVTSD2SI r32, xmm1/m64      | RM        | V/V                          | AVX                      | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.                  |
| VEX.128.F2.0F.W1 2D /r<br>VCVTSD2SI r64, xmm1/m64      | RM        | V/N.E. <sup>1</sup>          | AVX                      | Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64. |
| EVEX.LIG.F2.0F.W0 2D /r<br>VCVTSD2SI r32, xmm1/m64{er} | T1F       | V/V                          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.                  |
| EVEX.LIG.F2.0F.W1 2D /r<br>VCVTSD2SI r64, xmm1/m64{er} | T1F       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64. |

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000\_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

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

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSD2SI (EVEX encoded version)**

```

IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN  DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE  DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI

```

**(V)CVTSD2SI**

```

IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTSD2SI int __mm_cvtsd_i32(__m128d);
VCVTSD2SI int __mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 __mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 __mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 __mm_cvtsd_si64(__m128d);
CVTSD2SI int __mm_cvtsd_si32(__m128d a)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

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

## VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer

| Opcode/<br>Instruction                                  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.LIG.F2.OF.W0 79 /r<br>VCVTSD2USI r32, xmm1/m64{er} | T1F       | V/V                          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.                  |
| EVEX.LIG.F2.OF.W1 79 /r<br>VCVTSD2USI r64, xmm1/m64{er} | T1F       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64. |

### NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

### Operation

#### VCVTSD2USI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

ELSE DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

FI

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2USI unsigned int \_\_mm\_cvtsd\_u32(\_\_m128d);

VCVTSD2USI unsigned int \_\_mm\_cvt\_roundsd\_u32(\_\_m128d, int r);

VCVTSD2USI unsigned \_\_int64 \_\_mm\_cvtsd\_u64(\_\_m128d);

VCVTSD2USI unsigned \_\_int64 \_\_mm\_cvt\_roundsd\_u64(\_\_m128d, int r);

### SIMD Floating-Point Exceptions

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.



## CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F2 0F 5A /r<br>CVTSD2SS xmm1, xmm2/m64                                       | RM         | V/V                          | SSE2                     | Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.   |
| VEX.NDS.128.F2.0F.WIG 5A /r<br>VCVTSD2SS xmm1, xmm2,<br>xmm3/m64             | RVM        | V/V                          | AVX                      | Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.                    |
| EVEX.NDS.LIG.F2.0F.W1 5A /r<br>VCVTSD2SS xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w) | VEX.vvvv      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w) | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Converts a double-precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand.

When the “convert-from” operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAX\_VL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSD2SS (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VCVTSD2SS (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**CVTSD2SS (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAX_VL-1:32] Unmodified *)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTSD2SS __m128 __mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 __mm_maskz_cvtsd_ss(__mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 __mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 __mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 __mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int r);
CVTSD2SS __m128 __mm_cvtsd_ss(__m128 a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CVTISI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| F2 0F 2A /r<br>CVTISI2SD xmm1, r32/m32                          | RM         | V/V                          | SSE2                     | Convert one signed doubleword integer from r32/m32 to one double-precision floating-point value in xmm1. |
| F2 REX.W 0F 2A /r<br>CVTISI2SD xmm1, r/m64                      | RM         | V/N.E.                       | SSE2                     | Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.     |
| VEX.NDS.128.F2.0F.W0 2A /r<br>VCVTISI2SD xmm1, xmm2, r/m32      | RVM        | V/V                          | AVX                      | Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.   |
| VEX.NDS.128.F2.0F.W1 2A /r<br>VCVTISI2SD xmm1, xmm2, r/m64      | RVM        | V/N.E. <sup>1</sup>          | AVX                      | Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.     |
| EVEX.NDS.LIG.F2.0F.W0 2A /r<br>VCVTISI2SD xmm1, xmm2, r/m32     | T1S        | V/V                          | AVX512F                  | Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.   |
| EVEX.NDS.LIG.F2.0F.W1 2A /r<br>VCVTISI2SD xmm1, xmm2, r/m64[er] | T1S        | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.     |

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w) | VEX.vvvv      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w) | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTISI2SD is encoded with VEX.L=0. Encoding VCVTISI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSI2SD (EVEX encoded version)**

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**VCVTSI2SD (VEX.128 encoded version)**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**CVTSI2SD**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[MAX\_VL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSI2SD \_\_m128d\_mm\_cvti32\_sd(\_\_m128d s, int a);

VCVTSI2SD \_\_m128d\_mm\_cvt\_roundi32\_sd(\_\_m128d s, int a, int r);

VCVTSI2SD \_\_m128d\_mm\_cvti64\_sd(\_\_m128d s, \_\_int64 a);

VCVTSI2SD \_\_m128d\_mm\_cvt\_roundi64\_sd(\_\_m128d s, \_\_int64 a, int r);

CVTSI2SD \_\_m128d\_mm\_cvtsi64\_sd(\_\_m128d s, \_\_int64 a);

CVTSI2SD \_\_m128d\_mm\_cvtsi32\_sd(\_\_m128d a, int b)

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3 if W1, else Type 5.

EVEX-encoded instructions, see Exceptions Type E3NF if W1, else Type E10NF.

## CVTSSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| F3 0F 2A /r<br>CVTSSI2SS xmm1, r/m32                            | RM         | V/V                          | SSE                      | Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1. |
| F3 REX.W 0F 2A /r<br>CVTSSI2SS xmm1, r/m64                      | RM         | V/N.E.                       | SSE                      | Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.   |
| VEX.NDS.128.F3.0F.W0 2A /r<br>VCVTSSI2SS xmm1, xmm2, r/m32      | RVM        | V/V                          | AVX                      | Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1. |
| VEX.NDS.128.F3.0F.W1 2A /r<br>VCVTSSI2SS xmm1, xmm2, r/m64      | RVM        | V/N.E. <sup>1</sup>          | AVX                      | Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.   |
| EVEX.NDS.LIG.F3.0F.W0 2A /r<br>VCVTSSI2SS xmm1, xmm2, r/m32{er} | T1S        | V/V                          | AVX512F                  | Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1. |
| EVEX.NDS.LIG.F3.0F.W1 2A /r<br>VCVTSSI2SS xmm1, xmm2, r/m64{er} | T1S        | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.   |

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w) | VEX.vvvv      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w) | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a single-precision floating-point value in the destination operand (first operand). The “convert-from” source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAX\_VL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: RM:converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSSI2SS is encoded with VEX.L=0. Encoding VCVTSSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSI2SS (EVEX encoded version)**

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

**VCVTSI2SS (VEX.128 encoded version)**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

**CVTSI2SS (128-bit Legacy SSE version)**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[MAX\_VL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSI2SS \_\_m128 \_mm\_cvtsi32\_ss(\_\_m128 s, int a);

VCVTSI2SS \_\_m128 \_mm\_cvt\_roundi32\_ss(\_\_m128 s, int a, int r);

VCVTSI2SS \_\_m128 \_mm\_cvtsi64\_ss(\_\_m128 s, \_\_int64 a);

VCVTSI2SS \_\_m128 \_mm\_cvt\_roundi64\_ss(\_\_m128 s, \_\_int64 a, int r);

CVTSI2SS \_\_m128 \_mm\_cvtsi64\_ss(\_\_m128 s, \_\_int64 a);

CVTSI2SS \_\_m128 \_mm\_cvtsi32\_ss(\_\_m128 a, int b);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| F3 0F 5A /r<br>CVTSS2SD xmm1, xmm2/m32  | RM         | V/V                          | SSE2                     | Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.   |
| VEX.NDS.128.F3.0F.WIG 5A /r<br>VCVTSS2SD xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | AVX                      | Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.                    |
| EVEX.NDS.LIG.F3.0F.WO 5A /r<br>VCVTSS2SD xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae} | T1S        | V/V                          | AVX512F                  | Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w) | VEX.vvvv      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w) | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Converts a single-precision floating-point value in the “convert-from” source operand to a double-precision floating-point value in the destination operand. When the “convert-from” source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### VCVTSS2SD (EVEX encoded version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] = 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**VCVTSS2SD (VEX.128 encoded version)**

DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0])  
DEST[127:64] ← SRC1[127:64]  
DEST[MAX\_VL-1:128] ← 0

**CVTSS2SD (128-bit Legacy SSE version)**

DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);  
DEST[MAX\_VL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SD \_\_m128d \_\_mm\_cvt\_roundss\_sd(\_\_m128d a, \_\_m128 b, int r);  
VCVTSS2SD \_\_m128d \_\_mm\_mask\_cvt\_roundss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b, int r);  
VCVTSS2SD \_\_m128d \_\_mm\_maskz\_cvt\_roundss\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128 a, int r);  
VCVTSS2SD \_\_m128d \_\_mm\_mask\_cvtss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b);  
VCVTSS2SD \_\_m128d \_\_mm\_maskz\_cvtss\_sd(\_\_mmask8 m, \_\_m128d a, \_\_m128 b);  
CVTSS2SD \_\_m128d \_\_mm\_cvtss\_sd(\_\_m128d a, \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.  
EVEX-encoded instructions, see Exceptions Type E3.



## CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

| Opcode/<br>Instruction                                 | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| F3 0F 2D /r<br>CVTSS2SI r32, xmm1/m32                  | RM        | V/V                          | SSE                      | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32. |
| F3 REX.W 0F 2D /r<br>CVTSS2SI r64, xmm1/m32            | RM        | V/N.E.                       | SSE                      | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.   |
| VEX.128.F3.0F.W0 2D /r<br>VCVTSS2SI r32, xmm1/m32      | RM        | V/V                          | AVX                      | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32. |
| VEX.128.F3.0F.W1 2D /r<br>VCVTSS2SI r64, xmm1/m32      | RM        | V/N.E. <sup>1</sup>          | AVX                      | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.   |
| EVEX.LIG.F3.0F.W0 2D /r<br>VCVTSS2SI r32, xmm1/m32{er} | T1F       | V/V                          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32. |
| EVEX.LIG.F3.0F.W1 2D /r<br>VCVTSS2SI r64, xmm1/m32{er} | T1F       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.   |

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSS2SI (EVEX encoded version)**

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

**(V)VCVTSS2SI (Legacy and VEX.128 encoded version)**

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SI int \_\_mm\_cvtss\_i32( \_\_m128 a);

VCVTSS2SI int \_\_mm\_cvt\_roundss\_i32( \_\_m128 a, int r);

VCVTSS2SI \_\_int64 \_\_mm\_cvtss\_i64( \_\_m128 a);

VCVTSS2SI \_\_int64 \_\_mm\_cvt\_roundss\_i64( \_\_m128 a, int r);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

| Opcode/<br>Instruction                                  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.LIG.F3.OF.W0 79 /r<br>VCVTSS2USI r32, xmm1/m32{er} | T1F       | V/V                          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32. |
| EVEX.LIG.F3.OF.W1 79 /r<br>VCVTSS2USI r64, xmm1/m32{er} | T1F       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.   |

### NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

### Operation

#### VCVTSS2USI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned \_\_mm\_cvtss\_u32( \_\_m128 a);

VCVTSS2USI unsigned \_\_mm\_cvt\_roundss\_u32( \_\_m128 a, int r);

VCVTSS2USI unsigned \_\_int64 \_\_mm\_cvtss\_u64( \_\_m128 a);

VCVTSS2USI unsigned \_\_int64 \_\_mm\_cvt\_roundss\_u64( \_\_m128 a, int r);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| 66 0F E6 /r<br>CVTTPD2DQ xmm1, xmm2/m128                                     | RM        | V/V                          | SSE2                     | Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.                                      |
| VEX.128.66.0F.WIG E6 /r<br>VCVTPD2DQ xmm1, xmm2/m128                         | RM        | V/V                          | AVX                      | Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.                                      |
| VEX.256.66.0F.WIG E6 /r<br>VCVTPD2DQ xmm1, ymm2/m256                         | RM        | V/V                          | AVX                      | Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.                                    |
| EVEX.128.66.0F.W1 E6 /r<br>VCVTPD2DQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.     |
| EVEX.256.66.0F.W1 E6 /r<br>VCVTPD2DQ xmm1 {k1}{z},<br>ymm2/m256/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.   |
| EVEX.512.66.0F.W1 E6 /r<br>VCVTPD2DQ ymm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | FV        | V/V                          | AVX512F                  | Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

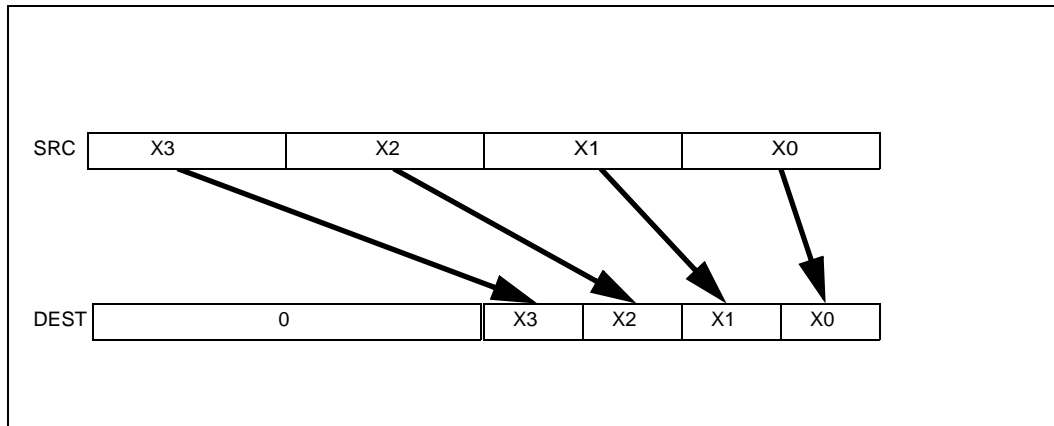


Figure 5-17. VCVTTPD2DQ (VEX.256 encoded version)

### Operation

**VCVTTPD2DQ (EVEX encoded versions) when src operand is a register**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTTPD2DQ (EVEX encoded versions) when src operand is a memory source**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
      FI
    FI

```

```

    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTTPD2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[MAX_VL-1:128] ← 0

```

**VCVTTPD2DQ (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAX_VL-1:64] ← 0

```

**CVTTPD2DQ (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ← 0
DEST[MAX_VL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2DQ __m256i __mm512_cvttpd_epi32( __m512d a);
VCVTTPD2DQ __m256i __mm512_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i __mm512_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i __mm512_cvtt_roundpd_epi32( __m512d a, int sae);
VCVTTPD2DQ __m256i __mm512_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m256i __mm512_maskz_cvtt_roundpd_epi32( __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i __mm256_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i __mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i __mm_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i __mm_maskz_cvttpd_epi32( __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i __mm256_cvttpd_epi32( __m256d src);
CVTTPD2DQ __m128i __mm_cvttpd_epi32( __m128d src);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

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

## VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F.W1 7A /r<br>VCVTTPD2QQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed double-precision floating-point values from zmm2/m128/m64bcst to two packed quadword integers in zmm1 using truncation with writemask k1.   |
| EVEX.256.66.0F.W1 7A /r<br>VCVTTPD2QQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1. |
| EVEX.512.66.0F.W1 7A /r<br>VCVTTPD2QQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | FV        | V/V                          | AVX512DQ                 | Convert eight packed double-precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.       |

### Instruction Operand Encoding

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

### Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

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

### Operation

**VCVTTPD2QQ (EVEX encoded version) when src operand is a register**

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger\_Truncate(SRC[i+63:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0



**VCVTTPD2QQ (EVEX encoded version) when src operand is a memory source**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2QQ __m512i __mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i __mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i __mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i __mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i __mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i __mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

| Opcode<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.0F.W1 78 /r<br>VCVTTPD2UDQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.     |
| EVEX.256.0F.W1 78 02 /r<br>VCVTTPD2UDQ xmm1 {k1}{z},<br>ymm2/m256/m64bcst   | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.   |
| EVEX.512.0F.W1 78 /r<br>VCVTTPD2UDQ ymm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | FV        | V/V                          | AVX512F                  | Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts with truncation packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask  $k1$ . The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

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

### Operation

**VCVTTPD2UDQ (EVEX encoded versions) when src2 operand is a register**

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

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 32$

$k \leftarrow j * 64$

  IF  $k1[j]$  OR \*no writemask\*

    THEN

      DEST[ $i+31:i$ ] ←

      Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[ $k+63:k$ ])

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE ; zeroing-masking

          DEST[ $i+31:i$ ] ← 0

    FI

FI;

```

ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
VCVTTPD2UDQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
          Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF;
  ENDIF;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTTPD2UDQ __m256i __mm512_cvttpd_epu32( __m512d a);
VCVTTPD2UDQ __m256i __mm512_mask_cvttpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i __mm512_maskz_cvttpd_epu32( __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i __mm512_cvtt_roundpd_epu32( __m512d a, int sae);
VCVTTPD2UDQ __m256i __mm512_mask_cvtt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i __mm512_maskz_cvtt_roundpd_epu32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i __mm256_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i __mm256_maskz_cvttpd_epu32( __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i __mm_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i __mm_maskz_cvttpd_epu32( __mmask8 k, __m128d a);

```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W1 78 /r<br>VCVTTPD2UQQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.   |
| EVEX.256.66.0F.W1 78 /r<br>VCVTTPD2UQQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1. |
| EVEX.512.66.0F.W1 78 /r<br>VCVTTPD2UQQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | FV        | V/V                          | AVX512DQ                 | Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.        |

### Instruction Operand Encoding

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

### Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTTPD2UQQ (EVEX encoded versions) when src operand is a register**

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

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+63:i] \leftarrow$

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger\_Truncate(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

$DEST[i+63:i] \leftarrow 0$

FI

FI;

ENDFOR

$DEST[MAX\_VL-1:VL] \leftarrow 0$

**VCVTTPD2UQQ (EVEX encoded versions) when src operand is a memory source**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
          Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2UQQ __mm<size>[_mask[z]]_cvttt[_round]pd_epu64
VCVTTPD2UQQ __m512i __mm512_cvttpd_epu64( __m512d a);
VCVTTPD2UQQ __m512i __mm512_mask_cvttpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_maskz_cvttpd_epu64( __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_cvtt_roundpd_epu64( __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_mask_cvtt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtt_roundpd_epu64( __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i __mm256_mask_cvttpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_maskz_cvttpd_epu64( __mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i __mm_mask_cvttpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __mm_maskz_cvttpd_epu64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| F3 0F 5B /r<br>CVTTPS2DQ xmm1, xmm2/m128                                       | RM        | V/V                          | SSE2                     | Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.  |
| VEX.128.F3.0F.WIG 5B /r<br>VCVTTPS2DQ xmm1, xmm2/m128                          | RM        | V/V                          | AVX                      | Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.  |
| VEX.256.F3.0F.WIG 5B /r<br>VCVTTPS2DQ ymm1, ymm2/m256                          | RM        | V/V                          | AVX                      | Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.                                      |
| EVEX.128.F3.0F.W0 5B /r<br>VCVTTPS2DQ xmm1 {k1}{z},<br>xmm2/m128/m32bcst       | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.       |
| EVEX.256.F3.0F.W0 5B /r<br>VCVTTPS2DQ ymm1 {k1}{z},<br>ymm2/m256/m32bcst       | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.     |
| EVEX.512.F3.0F.W0 5B /r<br>VCVTTPS2DQ zmm1 {k1}{z},<br>zmm2/m512/m32bcst {sae} | FV        | V/V                          | AVX512F                  | Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

**Operation****VCVTTPS2DQ (EVEX encoded versions) when src operand is a register**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTTPS2DQ (EVEX encoded versions) when src operand is a memory source**

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

```

FOR j ← 0 TO 15
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTTPS2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])

```

**VCVTTPS2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])  
 DEST[MAX\_VL-1:128] ← 0

**CVTTPS2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])  
 DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTPS2DQ \_\_m512i \_\_mm512\_cvttps\_epi32( \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_mask\_cvttps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvttps\_epi32( \_\_mmask16 k, \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_cvtt\_roundps\_epi32( \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_mask\_cvtt\_roundps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvtt\_roundps\_epi32( \_\_mmask16 k, \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_mask\_cvttps\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256 a);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_maskz\_cvttps\_epi32( \_\_mmask8 k, \_\_m256 a);  
 VCVTTPS2DQ \_\_m128i \_\_mm128\_mask\_cvttps\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128 a);  
 VCVTTPS2DQ \_\_m128i \_\_mm128\_maskz\_cvttps\_epi32( \_\_mmask8 k, \_\_m128 a);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_cvttps\_epi32( \_\_m256 a)  
 CVTTPS2DQ \_\_m128i \_\_mm128\_cvttps\_epi32( \_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

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



## VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.OF.W0 78 /r<br>VCVTTPS2UDQ xmm1 {k1}{z},<br>xmm2/m128/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.       |
| EVEX.256.OF.W0 78 /r<br>VCVTTPS2UDQ ymm1 {k1}{z},<br>ymm2/m256/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.     |
| EVEX.512.OF.W0 78 /r<br>VCVTTPS2UDQ zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae} | FV        | V/V                          | AVX512F                  | Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts with truncation packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

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

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

### Operation

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a register**

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ←

      Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[i+31:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2UDQ __m512i __mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i __mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i __mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i __mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i __mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i __mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i __mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i __mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W0 7A /r<br>VCVTTPS2QQ xmm1 {k1}{z},<br>xmm2/m64/m32bcst       | HV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.      |
| EVEX.256.66.0F.W0 7A /r<br>VCVTTPS2QQ ymm1 {k1}{z},<br>xmm2/m128/m32bcst      | HV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.   |
| EVEX.512.66.0F.W0 7A /r<br>VCVTTPS2QQ zmm1 {k1}{z},<br>ymm2/m256/m32bcst{sae} | HV        | V/V                          | AVX512DQ                 | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts with truncation packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

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

### Operation

#### VCVTTPS2QQ (EVEX encoded versions) when src operand is a register

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Single\_Precision\_To\_QuadInteger\_Truncate(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTTPS2QQ (EVEX encoded versions) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
          Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
  
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTTPS2QQ __m512i __mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i __mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_maskz_cvtt_roundps_epi64( __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i __mm256_mask_cvttps_epi64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i __mm256_maskz_cvttps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_mask_cvttps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_maskz_cvttps_epi64( __mmask8 k, __m128 a);
  
```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W0 78 /r<br>VCVTTPS2UQQ xmm1 {k1}{z},<br>xmm2/m64/m32bcst       | HV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 using truncation subject to writemask k1.      |
| EVEX.256.66.0F.W0 78 /r<br>VCVTTPS2UQQ ymm1 {k1}{z},<br>xmm2/m128/m32bcst      | HV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.   |
| EVEX.512.66.0F.W0 78 /r<br>VCVTTPS2UQQ zmm1 {k1}{z},<br>ymm2/m256/m32bcst{sae} | HV        | V/V                          | AVX512DQ                 | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts with truncation up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

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

### Operation

**VCVTTPS2UQQ (EVEX encoded versions) when src operand is a register**

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Single\_Precision\_To\_UQuadInteger\_Truncate(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTTPS2UQQ (EVEX encoded versions) when src operand is a memory source**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
          Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2UQQ __mm<size>[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i __mm512_cvttps_epu64( __m256 a);
VCVTTPS2UQQ __m512i __mm512_mask_cvttps_epu64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_maskz_cvttps_epu64( __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_cvtt_roundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_mask_cvtt_roundps_epu64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_maskz_cvtt_roundps_epu64( __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i __mm256_mask_cvttps_epu64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i __mm256_maskz_cvttps_epu64( __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_mask_cvttps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_maskz_cvttps_epu64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

## CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer

| Opcode/<br>Instruction                                   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F2 0F 2C /r<br>CVTTSD2SI r32, xmm1/m64                   | RM         | V/V                          | SSE2                     | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation. |
| F2 REX.W 0F 2C /r<br>CVTTSD2SI r64, xmm1/m64             | RM         | V/N.E.                       | SSE2                     | Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.   |
| VEX.128.F2.0F.W0 2C /r<br>VCVTTSD2SI r32, xmm1/m64       | RM         | V/V                          | AVX                      | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation. |
| VEX.128.F2.0F.W1 2C /r<br>VCVTTSD2SI r64, xmm1/m64       | T1F        | V/N.E. <sup>1</sup>          | AVX                      | Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.   |
| EVEX.LIG.F2.0F.W0 2C /r<br>VCVTTSD2SI r32, xmm1/m64{sae} | T1F        | V/V                          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation. |
| EVEX.LIG.F2.0F.W1 2C /r<br>VCVTTSD2SI r64, xmm1/m64{sae} | T1F        | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.   |

### NOTES:

- For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000\_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)VCVTTSD2SI (All versions)

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int \_\_mm\_cvttssd\_i32( \_\_m128d a);

VCVTTSD2SI int \_\_mm\_cvtt\_roundssd\_i32( \_\_m128d a, int sae);

VCVTTSD2SI \_\_int64 \_\_mm\_cvttssd\_i64( \_\_m128d a);

VCVTTSD2SI \_\_int64 \_\_mm\_cvtt\_roundssd\_i64( \_\_m128d a, int sae);

CVTTSD2SI int \_\_mm\_cvttssd\_si32( \_\_m128d a);

CVTTSD2SI \_\_int64 \_\_mm\_cvttssd\_si64( \_\_m128d a);

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.



## VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer

| Opcode/<br>Instruction                                    | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LIG.F2.OF.W0 78 /r<br>VCVTTSD2USI r32, xmm1/m64{sae} | T1F       | V/V                          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.                  |
| EVEX.LIG.F2.OF.W1 78 /r<br>VCVTTSD2USI r64, xmm1/m64{sae} | T1F       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation. |

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts with truncation a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

### Operation

#### VCVTTSD2USI (EVEX encoded version)

IF 64-Bit Mode and OperandSize = 64

```
THEN  DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
ELSE  DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
```

FI

### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2USI unsigned int _mm_cvttss_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundss_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttss_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundss_u64(__m128d, int sae);
```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer

| Opcode/<br>Instruction                                   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F3 0F 2C /r<br>CVTTSS2SI r32, xmm1/m32                   | RM         | V/V                          | SSE                      | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation. |
| F3 REX.W 0F 2C /r<br>CVTTSS2SI r64, xmm1/m32             | RM         | V/N.E.                       | SSE                      | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.   |
| VEX.128.F3.0F.W0 2C /r<br>VCVTTSS2SI r32, xmm1/m32       | RM         | V/V                          | AVX                      | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation. |
| VEX.128.F3.0F.W1 2C /r<br>VCVTTSS2SI r64, xmm1/m32       | RM         | V/N.E. <sup>1</sup>          | AVX                      | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.   |
| EVEX.LIG.F3.0F.W0 2C /r<br>VCVTTSS2SI r32, xmm1/m32{sae} | T1F        | V/V                          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation. |
| EVEX.LIG.F3.0F.W1 2C /r<br>VCVTTSS2SI r64, xmm1/m32{sae} | T1F        | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.   |

### NOTES:

- For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000\_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****(V)CVTTSS2SI (All versions)**

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTSS2SI int \_\_mm\_cvtss\_i32( \_\_m128 a);

VCVTTSS2SI int \_\_mm\_cvt\_roundss\_i32( \_\_m128 a, int sae);

VCVTTSS2SI \_\_int64 \_\_mm\_cvtss\_i64( \_\_m128 a);

VCVTTSS2SI \_\_int64 \_\_mm\_cvt\_roundss\_i64( \_\_m128 a, int sae);

CVTTSS2SI int \_\_mm\_cvtss\_si32( \_\_m128 a);

CVTTSS2SI \_\_int64 \_\_mm\_cvtss\_si64( \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

| Opcode/<br>Instruction                                    | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LIG.F3.OF.W0 78 /r<br>VCVTTSS2USI r32, xmm1/m32{sae} | T1F       | V/V                          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation. |
| EVEX.LIG.F3.OF.W1 78 /r<br>VCVTTSS2USI r64, xmm1/m32{sae} | T1F       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.   |

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1F   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts with truncation a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

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

### Operation

#### VCVTTSS2USI (EVEX encoded version)

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2USI unsigned int \_\_mm\_cvttss\_u32( \_\_m128 a);

VCVTTSS2USI unsigned int \_\_mm\_cvtt\_roundss\_u32( \_\_m128 a, int sae);

VCVTTSS2USI unsigned \_\_int64 \_\_mm\_cvttss\_u64( \_\_m128 a);

VCVTTSS2USI unsigned \_\_int64 \_\_mm\_cvtt\_roundss\_u64( \_\_m128 a, int sae);

### SIMD Floating-Point Exceptions

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F.W0 7A /r<br>VCVTUDQ2PD xmm1 {k1}{z},<br>xmm2/m64/m32bcst  | HV        | V/V                          | AVX512VL<br>AVX512F      | Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.          |
| EVEX.256.F3.0F.W0 7A /r<br>VCVTUDQ2PD ymm1 {k1}{z},<br>xmm2/m128/m32bcst | HV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.        |
| EVEX.512.F3.0F.W0 7A /r<br>VCVTUDQ2PD zmm1 {k1}{z},<br>ymm2/m256/m32bcst | HV        | V/V                          | AVX512F                  | Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

#### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

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

#### Operation

##### VCVTUDQ2PD (EVEX encoded versions) when src operand is a register

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

FOR j ← 0 TO KL-1

    i ← j \* 64

    k ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ←

            Convert\_UIInteger\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

    ELSE

        IF \*merging-masking\* ; merging-masking

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

        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTUDQ2PD (EVEX encoded versions) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          Convert_UIInteger_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_UIInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUDQ2PD __m512d __mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d __mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d __mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d __mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d __mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d __mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d __mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E5.

#UD If EVEX.vvvv != 1111B.

## VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.F2.0F.W0 7A /r<br>VCVTUDQ2PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single-precision floating-point values in xmm1 with writemask k1.            |
| EVEX.256.F2.0F.W0 7A /r<br>VCVTUDQ2PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single-precision floating-point values in zmm1 with writemask k1.           |
| EVEX.512.F2.0F.W0 7A /r<br>VCVTUDQ2PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTUDQ2PS (EVEX encoded version) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_UIInteger\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;



```
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source

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

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
          Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PS __m512 __mm512_cvtepu32_ps(__m512i a);
VCVTUDQ2PS __m512 __mm512_mask_cvtepu32_ps(__m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps(__mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_cvt_roundepu32_ps(__m512i a, int r);
VCVTUDQ2PS __m512 __mm512_mask_cvt_roundepu32_ps(__m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_maskz_cvt_roundepu32_ps(__mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 __mm256_cvtepu32_ps(__m256i a);
VCVTUDQ2PS __m256 __mm256_mask_cvtepu32_ps(__m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 __mm256_maskz_cvtepu32_ps(__mmask8 k, __m256i a);
VCVTUDQ2PS __m128 __mm_cvtepu32_ps(__m128i a);
VCVTUDQ2PS __m128 __mm_mask_cvtepu32_ps(__m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 __mm_maskz_cvtepu32_ps(__mmask8 k, __m128i a);
```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F.W1 7A /r<br>VCVTUQQ2PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double-precision floating-point values in xmm1 with writemask k1.     |
| EVEX.256.F3.0F.W1 7A /r<br>VCVTUQQ2PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.        |
| EVEX.512.F3.0F.W1 7A /r<br>VCVTUQQ2PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV         | V/V                          | AVX512DQ                 | Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTUQQ2PD (EVEX encoded version) when src operand is a register

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_UQuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

```

ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### VCVTUQQ2PD (EVEX encoded version) when src operand is a memory source

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
          Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+63:i] ←
          Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[j+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUQQ2PD __m512d __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d __mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d __mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d __mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d __mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d __mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d __mm_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.F2.0F.W1 7A /r<br>VCVTUQQ2PS xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.         |
| EVEX.256.F2.0F.W1 7A /r<br>VCVTUQQ2PS xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.        |
| EVEX.512.F2.0F.W1 7A /r<br>VCVTUQQ2PS ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV        | V/V                          | AVX512DQ                 | Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

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

### Operation

**VCVTUQQ2PS (EVEX encoded version) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_UQuadInteger\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

```

ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
VCVTUQQ2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
          Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUQQ2PS __m256 __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 __mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m128 __mm256_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PS __m128 __mm_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.F2.0F.W0 7B /r<br>VCVTUSI2SD xmm1, xmm2, r/m32     | T1S       | V/V                          | AVX512F                  | Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1. |
| EVEX.NDS.LIG.F2.0F.W1 7B /r<br>VCVTUSI2SD xmm1, xmm2, r/m64{er} | T1S       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.   |

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

### Operation

#### VCVTUSI2SD (EVEX encoded version)

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SD \_\_m128d \_mm\_cvту32\_sd( \_\_m128d s, unsigned a);

VCVTUSI2SD \_\_m128d \_mm\_cvту64\_sd( \_\_m128d s, unsigned \_\_int64 a);

VCVTUSI2SD \_\_m128d \_mm\_cvt\_roundu64\_sd( \_\_m128d s, unsigned \_\_int64 a, int r);

### **SIMD Floating-Point Exceptions**

Precision

### **Other Exceptions**

See Exceptions Type E3NF if W1, else type E10NF.

## VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.F3.0F.W0 7B /r<br>VCVTUSI2SS xmm1, xmm2, r/m32{er} | T1S       | V/V                          | AVX512F                  | Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1. |
| EVEX.NDS.LIG.F3.0F.W1 7B /r<br>VCVTUSI2SS xmm1, xmm2, r/m64{er} | T1S       | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.   |

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (w) | VEX.vvvv  | ModRM:r/m (r) | NA        |

### Description

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

### Operation

#### VCVTUSI2SS (EVEX encoded version)

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS \_\_m128 \_\_mm\_cvts32\_ss( \_\_m128 s, unsigned a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_roundu32\_ss( \_\_m128 s, unsigned a, int r);

VCVTUSI2SS \_\_m128 \_\_mm\_cvts64\_ss( \_\_m128 s, unsigned \_\_int64 a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_roundu64\_ss( \_\_m128 s, unsigned \_\_int64 a, int r);



## **SIMD Floating-Point Exceptions**

Precision

## **Other Exceptions**

See Exceptions Type E3NF.

## VDBPSADBw—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F3A.W0 42 /r ib<br>VDBPSADBw xmm1 {k1}{z}, xmm2,<br>xmm3/m128, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1. |
| EVEX.NDS.256.66.0F3A.W0 42 /r ib<br>VDBPSADBw ymm1 {k1}{z}, ymm2,<br>ymm3/m256, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1. |
| EVEX.NDS.512.66.0F3A.W0 42 /r ib<br>VDBPSADBw zmm1 {k1}{z}, zmm2,<br>zmm3/m512, imm8 | FVM       | V/V                          | AVX512BW                 | Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see "Tmp1" in Figure 5-18, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.
- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2 and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bit lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

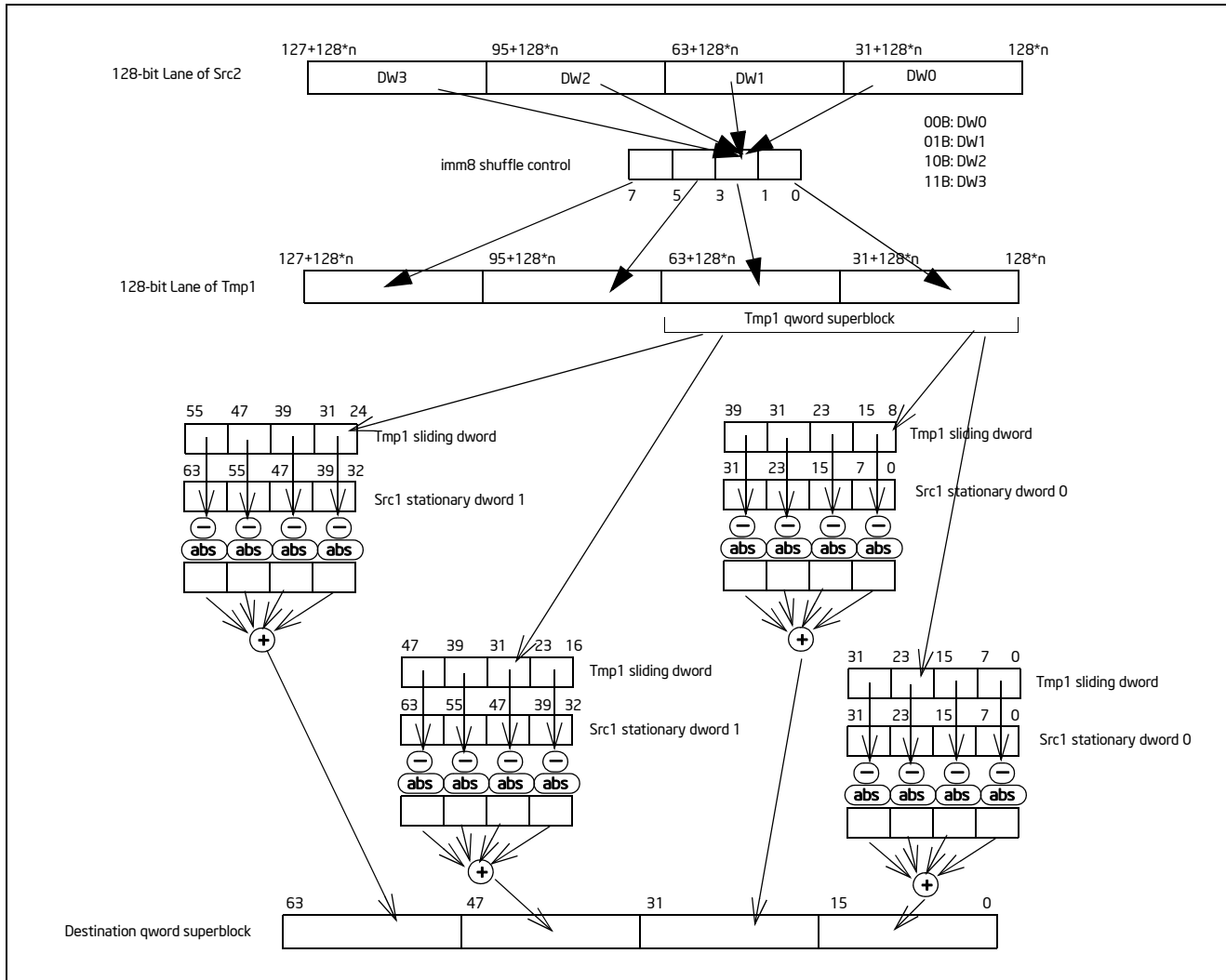


Figure 5-18. 64-bit Super Block of SAD Operation in VDBPSADBW

### Operation

#### VDBPSADBW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

Selection of quadruplets:

FOR I = 0 to VL step 128

TMP1[I+31:I] ← select (SRC2[I+127:I], imm8[1:0])

TMP1[I+63:I+32] ← select (SRC2[I+127:I], imm8[3:2])

TMP1[I+95:I+64] ← select (SRC2[I+127:I], imm8[5:4])

TMP1[I+127:I+96] ← select (SRC2[I+127:I], imm8[7:6])

END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64

TMP\_DEST[I+15:I] ← ABS(SRC1[I+7:I] - TMP1[I+7:I]) +  
ABS(SRC1[I+15:I+8] - TMP1[I+15:I+8]) +

```
ABS(SRC1[I+23: I+16]- TMP1[I+23: I+16]) +
ABS(SRC1[I+31: I+24]- TMP1[I+31: I+24])
```

```
TMP_DEST[I+31: I+16] ← ABS(SRC1[I+7: I] - TMP1[I+15: I+8]) +
ABS(SRC1[I+15: I+8]- TMP1[I+23: I+16]) +
ABS(SRC1[I+23: I+16]- TMP1[I+31: I+24]) +
ABS(SRC1[I+31: I+24]- TMP1[I+39: I+32])
TMP_DEST[I+47: I+32] ← ABS(SRC1[I+39: I+32] - TMP1[I+23: I+16]) +
ABS(SRC1[I+47: I+40]- TMP1[I+31: I+24]) +
ABS(SRC1[I+55: I+48]- TMP1[I+39: I+32]) +
ABS(SRC1[I+63: I+56]- TMP1[I+47: I+40])
```

```
TMP_DEST[I+63: I+48] ← ABS(SRC1[I+39: I+32] - TMP1[I+31: I+24]) +
ABS(SRC1[I+47: I+40] - TMP1[I+39: I+32]) +
ABS(SRC1[I+55: I+48] - TMP1[I+47: I+40]) +
ABS(SRC1[I+63: I+56] - TMP1[I+55: I+48])
```

```
ENDFOR
```

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 16
```

```
  IF k1[j] OR *no writemask*
```

```
    THEN DEST[i+15:i] ← TMP_DEST[j+15:i]
```

```
  ELSE
```

```
    IF *merging-masking* ; merging-masking
```

```
      THEN *DEST[i+15:i] remains unchanged*
```

```
    ELSE ; zeroing-masking
```

```
      DEST[i+15:i] ← 0
```

```
  FI
```

```
FI;
```

```
ENDFOR
```

```
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m128i _mm128i_dbsad_epu8(__m128i a, __m128i b);
VDBPSADBW __m128i _mm128i_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b);
VDBPSADBW __m128i _mm128i_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4NF.nb.

## VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 88 /r<br>VEXPANDPD xmm1 {k1}{z},<br>xmm2/m128 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 88 /r<br>VEXPANDPD ymm1 {k1}{z}, ymm2/m256    | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 88 /r<br>VEXPANDPD zmm1 {k1}{z}, zmm2/m512    | T1S       | V/V                          | AVX512F                  | Expand packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Expand (load) up to 8/4/2, contiguous, double-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VEXPANDPD (EVEX encoded versions)

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

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+63:i] ← SRC[k+63:k];

      k ← k + 64

    ELSE

      IF \*merging-masking\* ; merging-masking

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

      ELSE ; zeroing-masking

        THEN DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDPD __m512d __mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d __mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d __mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_mask_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d __mm_maskz_expandloadu_pd( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 88 /r<br>VEXPANDPS xmm1 {k1}{z}, xmm2/m128 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 88 /r<br>VEXPANDPS ymm1 {k1}{z}, ymm2/m256 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 88 /r<br>VEXPANDPS zmm1 {k1}{z}, zmm2/m512 | T1S       | V/V                          | AVX512F                  | Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Expand (load) up to 16/8/4, contiguous, single-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VEXPANDPS (EVEX encoded versions)

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

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+31:i] ← SRC[k+31:k];

      k ← k + 32

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE ; zeroing-masking

          DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDPS __m512 __mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 __mm_maskz_expandloadu_ps( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.



## VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| VEX.256.66.0F3A.W0 19 /r ib<br>VEXTRACTF128 xmm1/m128, ymm2,<br>imm8           | RMI       | V/V                          | AVX                      | Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.  |
| EVEX.256.66.0F3A.W0 19 /r ib<br>VEXTRACTF32X4 xmm1/m128 {k1}{z},<br>ymm2, imm8 | T4        | V/V                          | AVX512VL<br>AVX512F      | Extract 128 bits of packed single-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W0 19 /r ib<br>VEXTRACTF32x4 xmm1/m128 {k1}{z},<br>zmm2, imm8 | T4        | V/V                          | AVX512F                  | Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.256.66.0F3A.W1 19 /r ib<br>VEXTRACTF64X2 xmm1/m128 {k1}{z},<br>ymm2, imm8 | T2        | V/V                          | AVX512VL<br>AVX512DQ     | Extract 128 bits of packed double-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 19 /r ib<br>VEXTRACTF64X2 xmm1/m128 {k1}{z},<br>zmm2, imm8 | T2        | V/V                          | AVX512DQ                 | Extract 128 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W0 1B /r ib<br>VEXTRACTF32X8 ymm1/m256 {k1}{z},<br>zmm2, imm8 | T8        | V/V                          | AVX512DQ                 | Extract 256 bits of packed single-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 1B /r ib<br>VEXTRACTF64x4 ymm1/m256 {k1}{z},<br>zmm2, imm8 | T4        | V/V                          | AVX512F                  | Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En      | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|------------|---------------|---------------|-----------|-----------|
| RMI        | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |
| T2, T4, T8 | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |

### Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single-precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double-precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

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

The high 6 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

#### VEXTRACTF32x4 (EVEX encoded versions) when destination is a register

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:128] ← 0

#### VEXTRACTF32x4 (EVEX encoded versions) when destination is memory

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

```

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

#### **VEXTRACTF64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

```

FI;

FOR j ← 0 TO 1

i ← j \* 64

```

IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking*      ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking*    ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI

```

FI;

ENDFOR

DEST[MAX\_VL-1:128] ← 0

#### **VEXTRACTF64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

```

```

FI;

FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VEEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register**

```

VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

```

```

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*  ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0

```

**VEEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VEEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register**

```

VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

```

```

FOR j ← 0 TO 3
  i ← j * 64

```

```

IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0

```

**VEEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

```

```

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE ; merging-masking
      *DEST[i+63:i] remains unchanged*
  FI;
ENDFOR

```

**VEEXTRACTF128 (memory destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

```

**VEEXTRACTF128 (register destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEEXTRACTF32x4 __m128 __mm512_extractf32x4_ps(__m512 a, const int nidx);
VEEXTRACTF32x4 __m128 __mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEEXTRACTF32x4 __m128 __mm512_maskz_extractf32x4_ps( __mmask8 k, __m512 a, const int nidx);
VEEXTRACTF32x4 __m128 __mm256_extractf32x4_ps(__m256 a, const int nidx);
VEEXTRACTF32x4 __m128 __mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nidx);
VEEXTRACTF32x4 __m128 __mm256_maskz_extractf32x4_ps( __mmask8 k, __m256 a, const int nidx);
VEEXTRACTF32x8 __m256 __mm512_extractf32x8_ps(__m512 a, const int nidx);
VEEXTRACTF32x8 __m256 __mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEEXTRACTF32x8 __m256 __mm512_maskz_extractf32x8_ps( __mmask8 k, __m512 a, const int nidx);
VEEXTRACTF64x2 __m128d __mm512_extractf64x2_pd(__m512d a, const int nidx);
VEEXTRACTF64x2 __m128d __mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEEXTRACTF64x2 __m128d __mm512_maskz_extractf64x2_pd( __mmask8 k, __m512d a, const int nidx);
VEEXTRACTF64x2 __m128d __mm256_extractf64x2_pd(__m256d a, const int nidx);

```

VEXTRACTF64x2 \_\_m128d \_\_mm256\_mask\_extractf64x2\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m256d a, const int nidx);  
VEXTRACTF64x2 \_\_m128d \_\_mm256\_maskz\_extractf64x2\_pd(\_\_mmask8 k, \_\_m256d a, const int nidx);  
VEXTRACTF64x4 \_\_m256d \_\_mm512\_extractf64x4\_pd(\_\_m512d a, const int nidx);  
VEXTRACTF64x4 \_\_m256d \_\_mm512\_mask\_extractf64x4\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m512d a, const int nidx);  
VEXTRACTF64x4 \_\_m256d \_\_mm512\_maskz\_extractf64x4\_pd(\_\_mmask8 k, \_\_m512d a, const int nidx);  
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

VEX-encoded instructions, see Exceptions Type 6;  
EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.  
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.256.66.0F3A.W0 39 /r ib<br>VEXTRACTI128 xmm1/m128, ymm2,<br>imm8           | RMI       | V/V                          | AVX2                     | Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.                                       |
| EVEX.256.66.0F3A.W0 39 /r ib<br>VEXTRACTI32X4 xmm1/m128 {k1}{z},<br>ymm2, imm8 | T4        | V/V                          | AVX512VL<br>AVX512F      | Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W0 39 /r ib<br>VEXTRACTI32x4 xmm1/m128 {k1}{z},<br>zmm2, imm8 | T4        | V/V                          | AVX512F                  | Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.256.66.0F3A.W1 39 /r ib<br>VEXTRACTI64X2 xmm1/m128 {k1}{z},<br>ymm2, imm8 | T2        | V/V                          | AVX512VL<br>AVX512DQ     | Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.   |
| EVEX.512.66.0F3A.W1 39 /r ib<br>VEXTRACTI64X2 xmm1/m128 {k1}{z},<br>zmm2, imm8 | T2        | V/V                          | AVX512DQ                 | Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.   |
| EVEX.512.66.0F3A.W0 3B /r ib<br>VEXTRACTI32X8 ymm1/m256 {k1}{z},<br>zmm2, imm8 | T8        | V/V                          | AVX512DQ                 | Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 3B /r ib<br>VEXTRACTI64x4 ymm1/m256 {k1}{z},<br>zmm2, imm8 | T4        | V/V                          | AVX512F                  | Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.   |

### Instruction Operand Encoding

| Op/En      | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|------------|---------------|---------------|-----------|-----------|
| RMI        | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |
| T2, T4, T8 | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |

### Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

**VEXTRACTI32x8:** The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

**VEXTRACTI64x4:** The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

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

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### **VEXTRACTI32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

```
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
```

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[j+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:128] ← 0

#### **VEXTRACTI32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

```
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
```



```

    11: TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.
FI;

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VEXTRACTI64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

```

FI;

FOR j ← 0 TO 1

```

  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*  ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI;

```

ENDFOR

```

DEST[MAX_VL-1:128] ← 0

```

**VEXTRACTI64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]

```

```

01: TMP_DEST[127:0] ← SRC1[255:128]
10: TMP_DEST[127:0] ← SRC1[383:256]
11: TMP_DEST[127:0] ← SRC1[511:384]

```

```
ESAC.
```

```
FI;
```

```
FOR j ← 0 TO 1
```

```
  i ← j * 64
```

```
  IF k1[j] OR *no writemask*
```

```
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
```

```
    ELSE *DEST[i+63:i] remains unchanged*       ; merging-masking
```

```
  FI;
```

```
ENDFOR
```

#### **VEEXTRACT132x8 (EVEX.U1.512 encoded version) when destination is a register**

```
VL = 512
```

```
CASE (imm8[0]) OF
```

```
  0: TMP_DEST[255:0] ← SRC1[255:0]
```

```
  1: TMP_DEST[255:0] ← SRC1[511:256]
```

```
ESAC.
```

```
FOR j ← 0 TO 7
```

```
  i ← j * 32
```

```
  IF k1[j] OR *no writemask*
```

```
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
```

```
    ELSE
```

```
      IF *merging-masking*                       ; merging-masking
```

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

```
        ELSE *zeroing-masking*                 ; zeroing-masking
```

```
          DEST[i+31:i] ← 0
```

```
    FI
```

```
  FI;
```

```
ENDFOR
```

```
DEST[MAX_VL-1:256] ← 0
```

#### **VEEXTRACT132x8 (EVEX.U1.512 encoded version) when destination is memory**

```
CASE (imm8[0]) OF
```

```
  0: TMP_DEST[255:0] ← SRC1[255:0]
```

```
  1: TMP_DEST[255:0] ← SRC1[511:256]
```

```
ESAC.
```

```
FOR j ← 0 TO 7
```

```
  i ← j * 32
```

```
  IF k1[j] OR *no writemask*
```

```
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
```

```
    ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
```

```
  FI;
```

```
ENDFOR
```

#### **VEEXTRACT164x4 (EVEX.512 encoded version) when destination is a register**

```
VL = 512
```

```
CASE (imm8[0]) OF
```

```
  0: TMP_DEST[255:0] ← SRC1[255:0]
```

```
  1: TMP_DEST[255:0] ← SRC1[511:256]
```

ESAC.

```

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0

```

#### **VEEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.
FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

#### **VEEXTRACTI128 (memory destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

```

#### **VEEXTRACTI128 (register destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAX_VL-1:128] ← 0

```

#### **Intel C/C++ Compiler Intrinsic Equivalent**

```

VEEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nid);
VEEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nid);
VEEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nid);
VEEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nid);
VEEXTRACTI32x8 __m256i _mm512_extracti32x8_epi32(__m512i a, const int nid);
VEEXTRACTI32x8 __m256i _mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nid);
VEEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nid);

```

VEXTRACTI64x2 \_\_m128i \_\_mm512\_maskz\_extracti64x2\_epi64( \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i \_\_mm256\_extracti64x2\_epi64(\_\_m256i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i \_\_mm256\_mask\_extracti64x2\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m256i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i \_\_mm256\_maskz\_extracti64x2\_epi64( \_\_mmask8 k, \_\_m256i a, const int nidx);  
 VEXTRACTI64x4 \_\_m256i \_\_mm512\_extracti64x4\_epi64(\_\_m512i a, const int nidx);  
 VEXTRACTI64x4 \_\_m256i \_\_mm512\_mask\_extracti64x4\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI64x4 \_\_m256i \_\_mm512\_maskz\_extracti64x4\_epi64( \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI128 \_\_m128i \_\_mm256\_extracti128\_si256(\_\_m256i a, int offset);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

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

## EXTRACTPS—Extract Packed Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| 66 0F 3A 17 /r ib<br>EXTRACTPS reg/m32, xmm1, imm8              | RMI       | VV                           | SSE4_1                   | Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable. |
| VEX.128.66.0F3A.WIG 17 /r ib<br>VEXTRACTPS reg/m32, xmm1, imm8  | RMI       | V/V                          | AVX                      | Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable. |
| EVEX.128.66.0F3A.WIG 17 /r ib<br>VEXTRACTPS reg/m32, xmm1, imm8 | T1S       | V/V                          | AVX512F                  | Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RMI   | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |
| T1S   | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |

### Description

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

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

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

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

### Operation

#### VEXTRACTPS (EVEX and VEX.128 encoded version)

SRC\_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

FI

**EXTRACTPS (128-bit Legacy SSE version)**

SRC\_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

EXTRACTPS int \_mm\_extract\_ps (\_\_m128 a, const int nidx);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 5; Additionally

EVEX-encoded instructions, see Exceptions Type E9NF.

#UD IF VEX.L = 0.

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

## VFIXUPIMMPD—Fix Up Special Packed Float64 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F3A.W1 54 /r ib<br>VFIXUPIMMPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512F      | Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.              |
| EVEX.NDS.256.66.0F3A.W1 54 /r ib<br>VFIXUPIMMPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512F      | Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.              |
| EVEX.NDS.512.66.0F3A.W1 54 /r ib<br>VFIXUPIMMPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{sae}, imm8 | FV        | V/V                          | AVX512F                  | Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2 | Operand 3     | Operand 4 |
|-------|------------------|-----------|---------------|-----------|
| FV    | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMPD` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports `#ZE` and `#IE` fault reporting (see details below).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.

**Operation**

```
enum TOKEN_TYPE
```

```
{
  QNAN_TOKEN ← 0,
  SNAN_TOKEN ← 1,
  ZERO_VALUE_TOKEN ← 2,
  POS_ONE_VALUE_TOKEN ← 3,
  NEG_INF_TOKEN ← 4,
  POS_INF_TOKEN ← 5,
  NEG_VALUE_TOKEN ← 6,
  POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]){
  tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)
```

; The required response from src3 table is extracted  
 token\_response[3:0] = tbl3[3+4\*j;4\*j];

```
CASE(token_response[3:0]) {
  0000: dest[63:0] ← dest[63:0]; ; preserve content of DEST
  0001: dest[63:0] ← tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[63:0] ← QNaN(tsrc[63:0]);
  0011: dest[63:0] ← QNaN_Indefinite;
  0100: dest[63:0] ← -INF;
  0101: dest[63:0] ← +INF;
  0110: dest[63:0] ← tsrc.sign? -INF : +INF;
  0111: dest[63:0] ← -0;
  1000: dest[63:0] ← +0;
  1001: dest[63:0] ← -1;
  1010: dest[63:0] ← +1;
  1011: dest[63:0] ← ½;
  1100: dest[63:0] ← 90.0;
  1101: dest[63:0] ← PI/2;
  1110: dest[63:0] ← MAX_FLOAT;
  1111: dest[63:0] ← -MAX_FLOAT;
} ; end of token_response CASE
```

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

IF (tsrc[63:0] of TOKEN\_TYPE: ZERO\_VALUE\_TOKEN) AND imm8[0] then set #ZE;

IF (tsrc[63:0] of TOKEN\_TYPE: ZERO\_VALUE\_TOKEN) AND imm8[1] then set #IE;

IF (tsrc[63:0] of TOKEN\_TYPE: ONE\_VALUE\_TOKEN) AND imm8[2] then set #ZE;



```

IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
    ; end fault reporting
return dest[63:0];
} ; end of FIXUPIMM_DP()

```

**VFIXUPIMMPD**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

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

THEN

DEST[i+63:i] ← FIXUPIMM\_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])

ELSE

DEST[i+63:i] ← FIXUPIMM\_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[j+63:i], imm8 [7:0])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

Immediate Control Description:

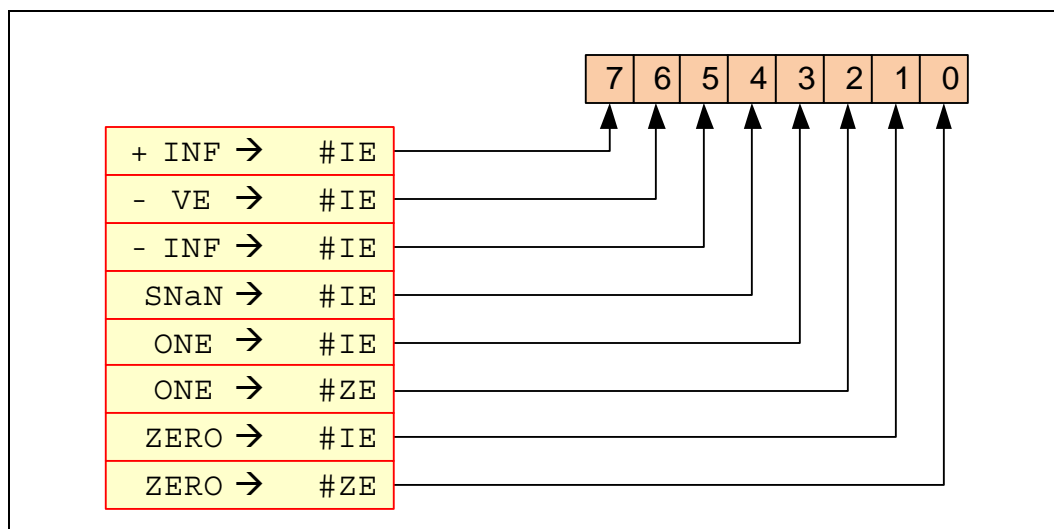


Figure 5-19. VFIXUPIMMPD Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMPD __m512d __mm512_fixupimm_pd( __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_maskz_fixupimm_pd( __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_fixupimm_round_pd( __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d __mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d __mm512_maskz_fixupimm_round_pd( __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d __mm256_fixupimm_pd( __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d __mm256_mask_fixupimm_pd(__m256d s, __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d __mm256_maskz_fixupimm_pd( __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m128d __mm_fixupimm_pd( __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d __mm_mask_fixupimm_pd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d __mm_maskz_fixupimm_pd( __mmask8 k, __m128d a, __m128i tbl, int imm);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E2.

## VFIXUPIMMPS—Fix Up Special Packed Float32 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F3A.W0 54 /r<br>VFIXUPIMMPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512F      | Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.              |
| EVEX.NDS.256.66.0F3A.W0 54 /r<br>VFIXUPIMMPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512F      | Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.              |
| EVEX.NDS.512.66.0F3A.W0 54 /r ib<br>VFIXUPIMMPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{saе}, imm8 | FV        | V/V                          | AVX512F                  | Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2 | Operand 3     | Operand 4 |
|-------|------------------|-----------|---------------|-----------|
| FV    | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMPS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports `#ZE` and `#IE` fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

**Operation**

enum TOKEN\_TYPE

```
{
  QNAN_TOKEN ← 0,
  SNAN_TOKEN ← 1,
  ZERO_VALUE_TOKEN ← 2,
  POS_ONE_VALUE_TOKEN ← 3,
  NEG_INF_TOKEN ← 4,
  POS_INF_TOKEN ← 5,
  NEG_VALUE_TOKEN ← 6,
  POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_SP ( dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
  tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)
```

; The required response from src3 table is extracted  
token\_response[3:0] = tbl3[3+4\*j;4\*j];

```
CASE(token_response[3:0]) {
  0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
  0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[31:0] ← QNaN(tsrc[31:0]);
  0011: dest[31:0] ← QNaN_Indefinite;
  0100: dest[31:0] ← -INF;
  0101: dest[31:0] ← +INF;
  0110: dest[31:0] ← tsrc.sign? -INF : +INF;
  0111: dest[31:0] ← -0;
  1000: dest[31:0] ← +0;
  1001: dest[31:0] ← -1;
  1010: dest[31:0] ← +1;
  1011: dest[31:0] ← ½;
  1100: dest[31:0] ← 90.0;
  1101: dest[31:0] ← PI/2;
  1110: dest[31:0] ← MAX_FLOAT;
  1111: dest[31:0] ← -MAX_FLOAT;
} ; end of token_response CASE
```

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

IF (tsrc[31:0] of TOKEN\_TYPE: ZERO\_VALUE\_TOKEN) AND imm8[0] then set #ZE;

IF (tsrc[31:0] of TOKEN\_TYPE: ZERO\_VALUE\_TOKEN) AND imm8[1] then set #IE;

IF (tsrc[31:0] of TOKEN\_TYPE: ONE\_VALUE\_TOKEN) AND imm8[2] then set #ZE;

```

IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
    ; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

```

**VFIXUPIMMPS (EVEX)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
        ELSE
          DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE DEST[i+31:i] ← 0 ; zeroing-masking
        FI
      FI;
  ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

Immediate Control Description:

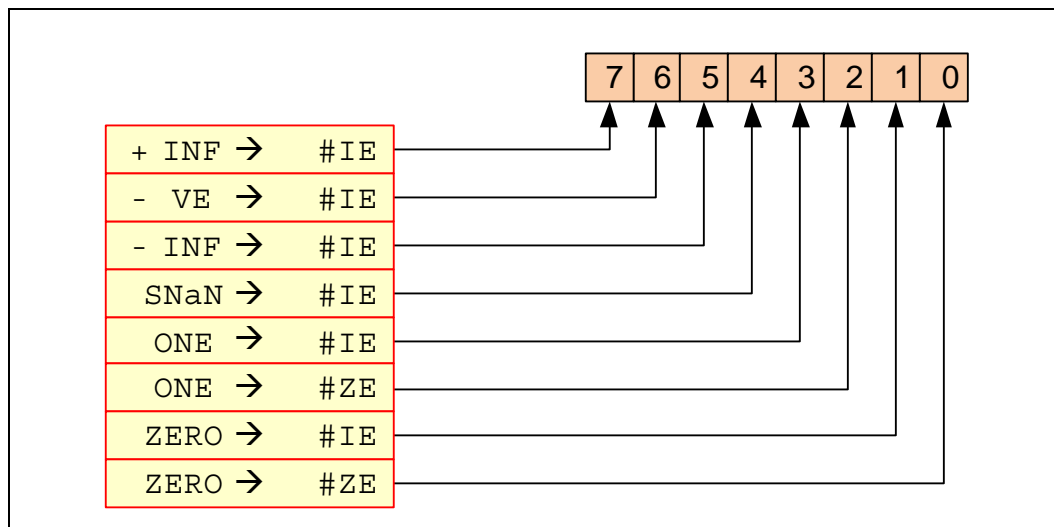


Figure 5-20. VFIXUPIMMPS Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMPS __m512 __mm512_fixupimm_ps( __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_fixupimm_round_ps( __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_round_ps( __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 __mm256_fixupimm_ps( __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_maskz_fixupimm_ps( __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 __mm_fixupimm_ps( __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i tbl, int imm);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E2.

## VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F3A.W1 55 /r ib<br>VFIXUPIMMSD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae}, imm8 | T1S       | V/V                          | AVX512F                  | Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2 | Operand 3     | Operand 4 |
|-------|------------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMSD` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

`Imm8` is used to set the required flags reporting. It supports `#ZE` and `#IE` fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the `imm8` bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

### Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
  tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)

```

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```

CASE(token_response[3:0]) {
  0000: dest[63:0] ← dest[63:0] ; preserve content of DEST
  0001: dest[63:0] ← tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[63:0] ← QNaN(tsrc[63:0]);
  0011: dest[63:0] ← QNaN_Indefinite;
  0100: dest[63:0] ← -INF;
  0101: dest[63:0] ← +INF;
  0110: dest[63:0] ← tsrc.sign? -INF : +INF;
  0111: dest[63:0] ← -0;
  1000: dest[63:0] ← +0;
  1001: dest[63:0] ← -1;
  1010: dest[63:0] ← +1;
  1011: dest[63:0] ← ½;
  1100: dest[63:0] ← 90.0;
  1101: dest[63:0] ← PI/2;
  1110: dest[63:0] ← MAX_FLOAT;
  1111: dest[63:0] ← -MAX_FLOAT;
} ; end of token_response CASE

```

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

```
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
```

```
 ; end fault reporting
```

```
return dest[63:0];
```

```
} ; end of FIXUPIMM_DP()
```



**VFIXUPIMMSD (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
  THEN DEST[63:0] ← FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
      ELSE DEST[63:0] ← 0 ; zeroing-masking
    FI
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

Immediate Control Description:

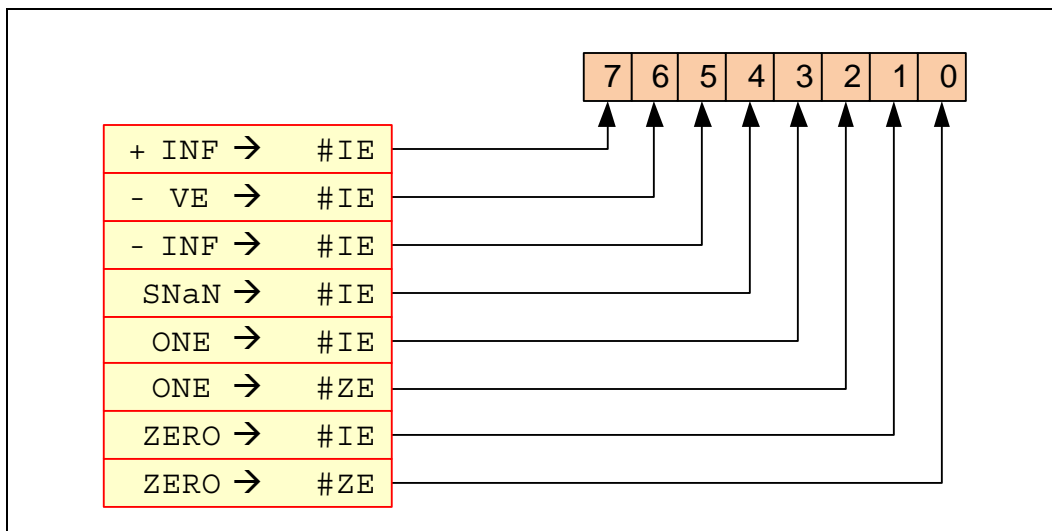


Figure 5-21. VFIXUPIMMSD Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMSD __m128d __mm_fixupimm_sd( __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __mm_mask_fixupimm_sd( __m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __mm_maskz_fixupimm_sd( __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __mm_fixupimm_round_sd( __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d __mm_mask_fixupimm_round_sd( __m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d __mm_maskz_fixupimm_round_sd( __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F3A.W0 55 /r ib<br>VFIXUPIMMSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae}, imm8 | T1S       | V/V                          | AVX512F                  | Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2 | Operand 3     | Operand 4 |
|-------|------------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMSS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

### Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```

FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
  tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)

; The required response from src3 table is extracted
token_response[3:0] = tbl3[3+4*j:4*j];

CASE(token_response[3:0]) {
  0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
  0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[31:0] ← QNaN(tsrc[31:0]);
  0011: dest[31:0] ← QNaN_Indefinite;
  0100: dest[31:0] ← -INF;
  0101: dest[31:0] ← +INF;
  0110: dest[31:0] ← tsrc.sign? -INF : +INF;
  0111: dest[31:0] ← -0;
  1000: dest[31:0] ← +0;
  1001: dest[31:0] ← -1;
  1010: dest[31:0] ← +1;
  1011: dest[31:0] ← ½;
  1100: dest[31:0] ← 90.0;
  1101: dest[31:0] ← PI/2;
  1110: dest[31:0] ← MAX_FLOAT;
  1111: dest[31:0] ← -MAX_FLOAT;
} ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

```

**VFIXUPIMMSS (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] ← FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE DEST[31:0] ← 0 ; zeroing-masking
    FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
    
```

Immediate Control Description:

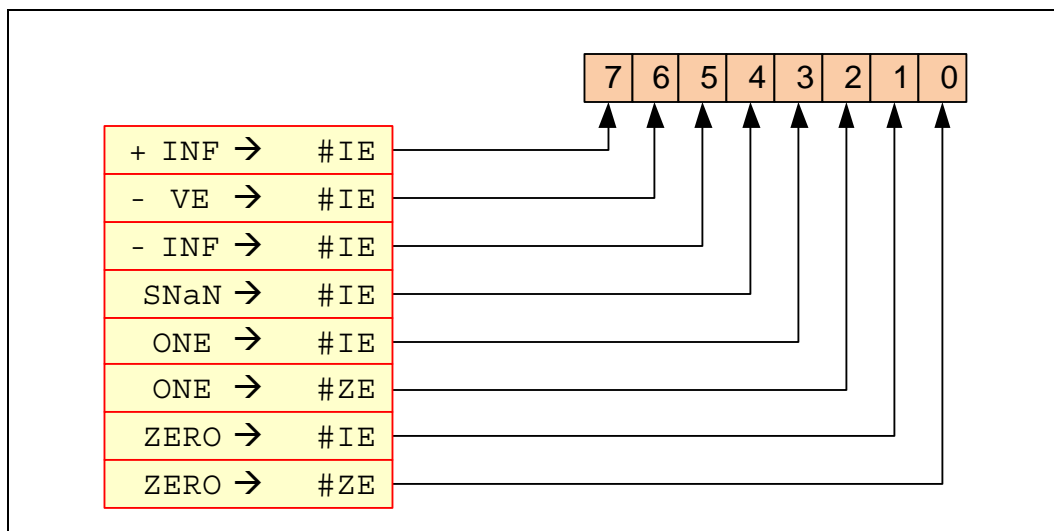


Figure 5-22. VFIXUPIMMSS Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMSS __m128 __mm_fixupimm_ss( __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_ss( __m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_ss( __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_fixupimm_round_ss( __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_round_ss( __m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_round_ss( __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
    
```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W1 98 /r<br>VFMADD132PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W1 A8 /r<br>VFMADD213PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W1 B8 /r<br>VFMADD231PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.          |
| VEX.NDS.256.66.0F38.W1 98 /r<br>VFMADD132PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W1 A8 /r<br>VFMADD213PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W1 B8 /r<br>VFMADD231PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.          |
| EVEX.NDS.128.66.0F38.W1 98 /r<br>VFMADD132PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | RVM       | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W1 A8 /r<br>VFMADD213PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W1 B8 /r<br>VFMADD231PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W1 98 /r<br>VFMADD132PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W1 A8 /r<br>VFMADD213PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W1 B8 /r<br>VFMADD231PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W1 98 /r<br>VFMADD132PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W1 A8 /r<br>VFMADD213PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W1 B8 /r<br>VFMADD231PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1. |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (r, w) | EVEX.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, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in `reg_field`. The second source operand is a ZMM register and encoded in `EVEX.vvvv`. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask `k1`.

**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.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

    MAXNUM ← 2

ELSEIF (VEX.256)

    MAXNUM ← 4

FI

For i = 0 to MAXNUM-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[MAX\_VL-1:128] ← 0

```

ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[j+63:i] + SRC2[j+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0

```

```

    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
  DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  ENDIF
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    ENDIF
  ENDIF
  DEST[MAX_VL-1:VL] ← 0

```



**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
        ELSE
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*

```

```

THEN
  IF (EVEX.b = 1)
    THEN
      DEST[i+63:i] ←
      RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
    ELSE
      DEST[i+63:i] ←
      RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[i+63:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDxxxPD __m512d __mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d __mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d __mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d __mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d __mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d __mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d __mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d __mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d __mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d __mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d __mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d __mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d __mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d __mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d __mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d __mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W0 98 /r<br>VFMADD132PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W0 A8 /r<br>VFMADD213PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W0 B8 /r<br>VFMADD231PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.          |
| VEX.NDS.256.66.0F38.W0 98 /r<br>VFMADD132PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W0 A8 /r<br>VFMADD213PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.          |
| VEX.NDS.256.66.0F38.0 B8 /r<br>VFMADD231PS ymm1, ymm2,<br>ymm3/m256                       | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.          |
| EVEX.NDS.128.66.0F38.W0 98 /r<br>VFMADD132PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W0 A8 /r<br>VFMADD213PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W0 B8 /r<br>VFMADD231PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W0 98 /r<br>VFMADD132PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W0 A8 /r<br>VFMADD213PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W0 B8 /r<br>VFMADD231PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W0 98 /r<br>VFMADD132PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W0 A8 /r<br>VFMADD213PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W0 B8 /r<br>VFMADD231PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1. |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (r, w) | EVEX.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, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in `reg_field`. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask `k1`.

**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.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN

    MAXNUM ← 4

ELSEIF (VEX.256)

    MAXNUM ← 8

FI

For i = 0 to MAXNUM-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[MAX\_VL-1:128] ← 0

```
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

#### **VFMADD213PS DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

#### **VFMADD231PS DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

#### **VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] + SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
```

```

    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(SRC2[i+31:i]*SRC3[j+31:i] + DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*

```

```

THEN
  IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
    ELSE
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| VEX.DDS.LIG.66.0F38.W1 99 /r<br>VFMADD132SD xmm1, xmm2,<br>xmm3/m64              | RVM       | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 A9 /r<br>VFMADD213SD xmm1, xmm2,<br>xmm3/m64              | RVM       | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 B9 /r<br>VFMADD231SD xmm1, xmm2,<br>xmm3/m64              | RVM       | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 99 /r<br>VFMADD132SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S       | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 A9 /r<br>VFMADD213SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S       | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 B9 /r<br>VFMADD231SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S       | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD multiply-add computation on the low 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 first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

**VFMADD132SD:** Multiplies the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point values in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD213SD:** Multiplies the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD231SD:** Multiplies the low double-precision floating-point value from the second source to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

### Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

#### VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

#### VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

```

```

IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
  FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← MAX_VL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d __mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.DDS.LIG.66.0F38.W0 99 /r<br>VFMADD132SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 A9 /r<br>VFMADD213SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 B9 /r<br>VFMADD231SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 99 /r<br>VFMADD132SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 A9 /r<br>VFMADD213SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 B9 /r<br>VFMADD231SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD multiply-add computation on 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 first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

**VFMADD132SS:** Multiplies the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VFMADD213SS:** Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VFMADD231SS:** Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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.

## Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

```
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
```

FI;

IF k1[0] or \*no writemask\*

```
THEN DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
        THEN DEST[31:0] ← 0
```

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

### VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

```
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
```

FI;

IF k1[0] or \*no writemask\*

```
THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
        THEN DEST[31:0] ← 0
```

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

**VFMAADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0]] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMAADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMAADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMAADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMAADDxxxSS __m128 __mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMAADDxxxSS __m128 __mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMAADDxxxSS __m128 __mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMAADDxxxSS __m128 __mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMAADDxxxSS __m128 __mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.DDS.128.66.0F38.W1 96 /r<br>VFMADDSUB132PD xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W1 A6 /r<br>VFMADDSUB213PD xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W1 B6 /r<br>VFMADDSUB231PD xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.                                  |
| VEX.DDS.256.66.0F38.W1 96 /r<br>VFMADDSUB132PD ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W1 A6 /r<br>VFMADDSUB213PD ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W1 B6 /r<br>VFMADDSUB231PD ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.                                  |
| EVEX.DDS.128.66.0F38.W1 A6 /r<br>VFMADDSUB213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W1 B6 /r<br>VFMADDSUB231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W1 96 /r<br>VFMADDSUB132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W1 A6 /r<br>VFMADDSUB213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W1 B6 /r<br>VFMADDSUB231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W1 96 /r<br>VFMADDSUB132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W1 A6 /r<br>VFMADDSUB213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|-----------------------------|--------------------------|---|
| EVEX.DDS.512.66.0F38.W1 B6 /r<br>VFMADDSUB231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV         | V/V                         | AVX512F                  | Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W1 96 /r<br>VFMADDSUB132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV         | V/V                         | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

VFMADDSUB132PD: Multiplies the two, four, or eight 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, four, or eight 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, four, or eight 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).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

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.



**Operation**

In the operations below, "\*" and "-" symbols represent multiplication and subtraction 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[MAX_VL-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[MAX_VL-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[MAX_VL-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
```

**VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

```

        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
    ELSE DEST[i+63:i] ←
        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
    FI
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
        THEN
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
        ELSE
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        FI;
    FI
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            ELSE DEST[i+63:i] ←
                RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                    ELSE
                        DEST[i+63:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[j+63:i] - DEST[j+63:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[j+63:i] ←
            RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[63:0] + DEST[j+63:i])
        ELSE
            DEST[j+63:i] ←
            RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[j+63:i] + DEST[j+63:i])
        FI;
    FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[j+63:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[j+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDSUBxxxPD __m512d __mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d __mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d __mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.DDS.128.66.0F38.W0 96 /r<br>VFMADDSUB132PS xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W0 A6 /r<br>VFMADDSUB213PS xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W0 B6 /r<br>VFMADDSUB231PS xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.                                  |
| VEX.DDS.256.66.0F38.W0 96 /r<br>VFMADDSUB132PS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W0 A6 /r<br>VFMADDSUB213PS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W0 B6 /r<br>VFMADDSUB231PS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.                                  |
| EVEX.DDS.128.66.0F38.W0 A6 /r<br>VFMADDSUB213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W0 B6 /r<br>VFMADDSUB231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W0 96 /r<br>VFMADDSUB132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W0 A6 /r<br>VFMADDSUB213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W0 B6 /r<br>VFMADDSUB231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W0 96 /r<br>VFMADDSUB132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W0 A6 /r<br>VFMADDSUB213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W0 B6 /r<br>VFMADDSUB231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W0 96 /r<br>VFMADDSUB132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1. |

## Instruction Operand Encoding

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

## Description

**VFMADDSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding 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 packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding 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 packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding 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 packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADDSUB213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADDSUB231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])

```



```

        ELSE DEST[j+31:i] ←
            RoundFPControl(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
    FI
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] - SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[j+31:i] ←
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] + SRC2[j+31:i])
                            ELSE
                                DEST[j+31:i] ←
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
                            FI;
                        FI
                    ELSE
                        IF *merging-masking*                ; merging-masking
                            THEN *DEST[j+31:i] remains unchanged*
                        ELSE                                ; zeroing-masking
                            DEST[j+31:i] ← 0
                        FI
                    FI;
                ELSE
                    IF *merging-masking*                ; merging-masking
                        THEN *DEST[j+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[j+31:i] ← 0
                    FI
                FI;
            ELSE
                DEST[MAX_VL-1:VL] ← 0
            ENDFOR

```

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

```

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

```

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
        ELSE
            DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    FI;
FI
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[i+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDSUBxxxPS __m512 __mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 __mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 __mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 __mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.DDS.128.66.0F38.W1 97 /r<br>VFMSUBADD132PD xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W1 A7 /r<br>VFMSUBADD213PD xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W1 B7 /r<br>VFMSUBADD231PD xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.                                  |
| VEX.DDS.256.66.0F38.W1 97 /r<br>VFMSUBADD132PD ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W1 A7 /r<br>VFMSUBADD213PD ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W1 B7 /r<br>VFMSUBADD231PD ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.                                  |
| EVEX.DDS.128.66.0F38.W1 97 /r<br>VFMSUBADD132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W1 A7 /r<br>VFMSUBADD213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W1 B7 /r<br>VFMSUBADD231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W1 97 /r<br>VFMSUBADD132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W1 A7 /r<br>VFMSUBADD213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W1 B7 /r<br>VFMSUBADD231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W1 97 /r<br>VFMSUBADD132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.DDS.512.66.0F38.W1 A7 /r<br>VFMSUBADD213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W1 B7 /r<br>VFMSUBADD231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV         | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

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

### Description

**VFMSUBADD132PD:** Multiplies the two, four, or eight 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, four, or eight 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, four, or eight 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).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

**Operation**

In the operations below, “\*” and “+” symbols represent multiplication and addition 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[MAX_VL-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[MAX_VL-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[MAX_VL-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
```

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

```

        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
    ELSE DEST[i+63:i] ←
        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
    FI
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
        THEN
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
        FI
    ELSE
        IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```



**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[j+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[j+63:i] ←
                    RoundFPControl(SRC2[j+63:i]*SRC3[j+63:i] + DEST[j+63:i])
            ELSE DEST[j+63:i] ←
                RoundFPControl(SRC2[j+63:i]*SRC3[j+63:i] - DEST[j+63:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[j+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[j+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+63:i] ←
                                RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[63:0] + DEST[j+63:i])
                    ELSE
                        DEST[j+63:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

        ELSE
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
    FI;
FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMSUBADDxxxPD __m512d __mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m256d __mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|-----------------------------|--------------------------|---|
| VEX.DDS.128.66.0F38.W0 97 /r<br>VFMSUBADD132PS xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                         | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W0 A7 /r<br>VFMSUBADD213PS xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                         | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.                                  |
| VEX.DDS.128.66.0F38.W0 B7 /r<br>VFMSUBADD231PS xmm1, xmm2,<br>xmm3/m128                      | RVM        | V/V                         | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.                                  |
| VEX.DDS.256.66.0F38.W0 97 /r<br>VFMSUBADD132PS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                         | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W0 A7 /r<br>VFMSUBADD213PS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                         | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.                                  |
| VEX.DDS.256.66.0F38.W0 B7 /r<br>VFMSUBADD231PS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                         | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.                                  |
| EVEX.DDS.128.66.0F38.W0 97 /r<br>VFMSUBADD132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV         | V/V                         | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W0 A7 /r<br>VFMSUBADD213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV         | V/V                         | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.128.66.0F38.W0 B7 /r<br>VFMSUBADD231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV         | V/V                         | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W0 97 /r<br>VFMSUBADD132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV         | V/V                         | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W0 A7 /r<br>VFMSUBADD213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV         | V/V                         | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.256.66.0F38.W0 B7 /r<br>VFMSUBADD231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV         | V/V                         | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W0 97 /r<br>VFMSUBADD132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV         | V/V                         | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W0 A7 /r<br>VFMSUBADD213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV         | V/V                         | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1. |
| EVEX.DDS.512.66.0F38.W0 B7 /r<br>VFMSUBADD231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV         | V/V                         | AVX512F                  | Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1. |

## Instruction Operand Encoding

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

## Description

**VFMSUBADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding 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 packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding 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 packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding 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 packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUBADD213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUBADD231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])

```

```

        ELSE DEST[j+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
    FI
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
                    FI;
                FI
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[j+31:i] ← 0
                FI
            FI;
        ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[31:0])

FI;



```

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

```

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            ELSE DEST[i+31:i] ←
                RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                    ELSE
                        DEST[i+31:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
                DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI;
        FI
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMSUBADDxxxPS __m512 __mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 __mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256 __mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 __mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 __mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 __mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 __mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 __mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W1 9A /r<br>VFMSUB132PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.                                  |
| VEX.NDS.128.66.0F38.W1 AA /r<br>VFMSUB213PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.                                  |
| VEX.NDS.128.66.0F38.W1 BA /r<br>VFMSUB231PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.                                  |
| VEX.NDS.256.66.0F38.W1 9A /r<br>VFMSUB132PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.                                  |
| VEX.NDS.256.66.0F38.W1 AA /r<br>VFMSUB213PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.                                  |
| VEX.NDS.256.66.0F38.W1 BA /r<br>VFMSUB231PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.S                                 |
| EVEX.NDS.128.66.0F38.W1 9A /r<br>VFMSUB132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.NDS.128.66.0F38.W1 AA /r<br>VFMSUB213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1. |
| EVEX.NDS.128.66.0F38.W1 BA /r<br>VFMSUB231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F38.W1 9A /r<br>VFMSUB132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F38.W1 AA /r<br>VFMSUB213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F38.W1 BA /r<br>VFMSUB231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F38.W1 9A /r<br>VFMSUB132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F38.W1 AA /r<br>VFMSUB213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F38.W1 BA /r<br>VFMSUB231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1. |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (r, w) | EVEX.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, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0

```

```
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

**VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)**

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

**VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)**

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[j+63:i] - SRC2[j+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
```

```

    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

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

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDIF
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
+31:i])
        ELSE
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
          FI;
        ELSE
          IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                           ; zeroing-masking
              DEST[i+63:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE                           ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxPD __m512d __mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d __mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d __mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d __mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d __mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/E<br>n | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEEX.NDS.128.66.0F38.W0 9A /r<br>VFMSUB132PS xmm1, xmm2,<br>xmm3/m128                     | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.          |
| VEEX.NDS.128.66.0F38.W0 AA /r<br>VFMSUB213PS xmm1, xmm2,<br>xmm3/m128                     | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.          |
| VEEX.NDS.128.66.0F38.W0 BA /r<br>VFMSUB231PS xmm1, xmm2,<br>xmm3/m128                     | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.          |
| VEEX.NDS.256.66.0F38.W0 9A /r<br>VFMSUB132PS ymm1, ymm2,<br>ymm3/m256                     | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.          |
| VEEX.NDS.256.66.0F38.W0 AA /r<br>VFMSUB213PS ymm1, ymm2,<br>ymm3/m256                     | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.          |
| VEEX.NDS.256.66.0F38.0 BA /r<br>VFMSUB231PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.          |
| EVEX.NDS.128.66.0F38.W0 9A /r<br>VFMSUB132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W0 AA /r<br>VFMSUB213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W0 BA /r<br>VFMSUB231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W0 9A /r<br>VFMSUB132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W0 AA /r<br>VFMSUB213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W0 BA /r<br>VFMSUB231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W0 9A /r<br>VFMSUB132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W0 AA /r<br>VFMSUB213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W0 BA /r<br>VFMSUB231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1. |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (r, w) | EVEX.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, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0

```

```

ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### **VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### **VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### **VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0

```

```

    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[j+31:i] ←
            RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] - SRC3[31:0])
        ELSE
          DEST[j+31:i] ←
            RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] - SRC3[j+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[j+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[j+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[j+31:i] ←
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[j+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*

```

```

THEN
  IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
    ELSE
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMSUBxxxPS __m512 __mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 __mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 __mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.DDS.LIG.66.0F38.W1 9B /r<br>VFMSUB132SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 AB /r<br>VFMSUB213SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 BB /r<br>VFMSUB231SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 9B /r<br>VFMSUB132SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 AB /r<br>VFMSUB213SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 BB /r<br>VFMSUB231SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.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-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit 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 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(DEST[63:0]\*SRC3[63:0] - SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

#### VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]\*DEST[63:0] - SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0



**VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxSD __m128d __mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d __mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d __mm_fmsub_sd(__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.DDS.LIG.66.0F38.W0 9B /r<br>VFMSUB132SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 AB /r<br>VFMSUB213SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 BB /r<br>VFMSUB231SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 9B /r<br>VFMSUB132SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 AB /r<br>VFMSUB213SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 BB /r<br>VFMSUB231SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.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-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit 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 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0
```

#### VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0
```

**VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxSS __m128 __mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 __mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 __mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 __mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 __mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 __mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 __mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 __mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38.W1 9C /r<br>VFMADD132PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W1 AC /r<br>VFMADD213PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W1 BC /r<br>VFMADD231PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.          |
| VEX.NDS.256.66.0F38.W1 9C /r<br>VFMADD132PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W1 AC /r<br>VFMADD213PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W1 BC /r<br>VFMADD231PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.          |
| EVEX.NDS.128.66.0F38.W1 9C /r<br>VFMADD132PD xmm0 {k1}{z},<br>xmm1, xmm2/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W1 AC /r<br>VFMADD213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W1 BC /r<br>VFMADD231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W1 9C /r<br>VFMADD132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W1 AC /r<br>VFMADD213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W1 BC /r<br>VFMADD231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W1 9C /r<br>VFMADD132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1. |

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.512.66.0F38.W1 AC /r<br>VFNMADD213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W1 BC /r<br>VFNMADD231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1. |

### Instruction Operand Encoding

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

### Description

VFNMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

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.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[63:0]) + SRC2[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[j+63:i]) + SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0



**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*SRC3[i+63:i]) + DEST[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[63:0]) + DEST[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[j+63:i]) + DEST[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADDxxxPD \_\_m512d \_\_mm512\_fmadd\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c);

VFMADDxxxPD \_\_m512d \_\_mm512\_fmadd\_round\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, int r);

VFMADDxxxPD \_\_m512d \_\_mm512\_mask\_fmadd\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512d b, \_\_m512d c);

VFMADDxxxPD \_\_m512d \_\_mm512\_maskz\_fmadd\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, \_\_m512d c);

VFMADDxxxPD \_\_m512d \_\_mm512\_mask3\_fmadd\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, \_\_mmask8 k);

VFMADDxxxPD \_\_m512d \_\_mm512\_mask\_fnmadd\_round\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512d b, \_\_m512d c, int r);  
 VFMADDxxxPD \_\_m512d \_\_mm512\_maskz\_fnmadd\_round\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, \_\_m512d c, int r);  
 VFMADDxxxPD \_\_m512d \_\_mm512\_mask3\_fnmadd\_round\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, \_\_mmask8 k, int r);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_mask\_fnmadd\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256d b, \_\_m256d c);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_maskz\_fnmadd\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b, \_\_m256d c);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_mask3\_fnmadd\_pd(\_\_m256d a, \_\_m256d b, \_\_m256d c, \_\_mmask8 k);  
 VFMADDxxxPD \_\_m128d \_\_mm\_mask\_fnmadd\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c);  
 VFMADDxxxPD \_\_m128d \_\_mm\_maskz\_fnmadd\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c);  
 VFMADDxxxPD \_\_m128d \_\_mm\_mask3\_fnmadd\_pd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k);  
 VFMADDxxxPD \_\_m128d \_\_mm\_fnmadd\_pd (\_\_m128d a, \_\_m128d b, \_\_m128d c);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_fnmadd\_pd (\_\_m256d a, \_\_m256d b, \_\_m256d c);

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38.W0 9C /r<br>VFNMADD132PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W0 AC /r<br>VFNMADD213PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W0 BC /r<br>VFNMADD231PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.          |
| VEX.NDS.256.66.0F38.W0 9C /r<br>VFNMADD132PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W0 AC /r<br>VFNMADD213PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.          |
| VEX.NDS.256.66.0F38.0 BC /r<br>VFNMADD231PS ymm1, ymm2,<br>ymm3/m256                       | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.          |
| EVEX.NDS.128.66.0F38.W0 9C /r<br>VFNMADD132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W0 AC /r<br>VFNMADD213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W0 BC /r<br>VFNMADD231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W0 9C /r<br>VFNMADD132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W0 AC /r<br>VFNMADD213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W0 BC /r<br>VFNMADD231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W0 9C /r<br>VFNMADD132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W0 AC /r<br>VFNMADD213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W0 BC /r<br>VFNMADD231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1. |

## Instruction Operand Encoding

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

## Description

**VFMADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
    RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

          ELSE
            DEST[i+31:i] ←
              RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
            FI;
        ELSE
          IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
          ELSE                             ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI;
      ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```



**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDxxxPS __m512 _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

```

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| VEX.DDS.LIG.66.0F38.W1 9D /r<br>VFMADD132SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 AD /r<br>VFMADD213SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 BD /r<br>VFMADD231SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 9D /r<br>VFMADD132SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 AD /r<br>VFMADD213SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 BD /r<br>VFMADD231SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

**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 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).

**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 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).

**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 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 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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.

## Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

### VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) + SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

### VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) + SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d __mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| VEX.DDS.LIG.66.0F38.W0 9D /r<br>VFMADD132SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 AD /r<br>VFMADD213SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 BD /r<br>VFMADD231SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 9D /r<br>VFMADD132SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 AD /r<br>VFMADD213SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 BD /r<br>VFMADD231SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

**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 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).

**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 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).

**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 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 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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.

### Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN  $DEST[31:0] \leftarrow \text{RoundFPControl}(-(\text{DEST}[31:0] * \text{SRC3}[31:0]) + \text{SRC2}[31:0])$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN  $DEST[31:0] \leftarrow 0$

FI;

FI;

$DEST[127:32] \leftarrow DEST[127:32]$

$DEST[\text{MAX\_VL}-1:128] \leftarrow 0$

#### VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN  $DEST[31:0] \leftarrow \text{RoundFPControl}(-(\text{SRC2}[31:0] * \text{DEST}[31:0]) + \text{SRC3}[31:0])$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN  $DEST[31:0] \leftarrow 0$

FI;

FI;

$DEST[127:32] \leftarrow DEST[127:32]$

$DEST[\text{MAX\_VL}-1:128] \leftarrow 0$

**VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNSUB132PD/VFNSUB213PD/VFNSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38.W1 9E /r<br>VFNSUB132PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W1 AE /r<br>VFNSUB213PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.          |
| VEX.NDS.128.66.0F38.W1 BE /r<br>VFNSUB231PD xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.          |
| VEX.NDS.256.66.0F38.W1 9E /r<br>VFNSUB132PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W1 AE /r<br>VFNSUB213PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.          |
| VEX.NDS.256.66.0F38.W1 BE /r<br>VFNSUB231PD ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.          |
| EVEX.NDS.128.66.0F38.W1 9E /r<br>VFNSUB132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W1 AE /r<br>VFNSUB213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1. |
| EVEX.NDS.128.66.0F38.W1 BE /r<br>VFNSUB231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W1 9E /r<br>VFNSUB132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W1 AE /r<br>VFNSUB213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1. |
| EVEX.NDS.256.66.0F38.W1 BE /r<br>VFNSUB231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W1 9E /r<br>VFNSUB132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W1 AE /r<br>VFNSUB213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1. |
| EVEX.NDS.512.66.0F38.W1 BE /r<br>VFNSUB231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1. |



## Instruction Operand Encoding

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

## Description

**VFNMSUB132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(DEST[j+63:i]*SRC3[j+63:i]) - SRC2[j+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[j+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i]
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i]
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNSUBxxxPD __m512d __mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNSUBxxxPD __m512d __mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNSUBxxxPD __m512d __mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNSUBxxxPD __m512d __mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNSUBxxxPD __m512d __mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNSUBxxxPD __m512d __mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNSUBxxxPD __m512d __mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNSUBxxxPD __m512d __mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNSUBxxxPD __m256d __mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNSUBxxxPD __m256d __mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNSUBxxxPD __m256d __mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNSUBxxxPD __m128d __mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNSUBxxxPD __m128d __mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNSUBxxxPD __m128d __mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNSUBxxxPD __m128d __mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNSUBxxxPD __m256d __mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W0 9E /r<br>VFNMSUB132PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.             |
| VEX.NDS.128.66.0F38.W0 AE /r<br>VFNMSUB213PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.             |
| VEX.NDS.128.66.0F38.W0 BE /r<br>VFNMSUB231PS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.             |
| VEX.NDS.256.66.0F38.W0 9E /r<br>VFNMSUB132PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.             |
| VEX.NDS.256.66.0F38.W0 AE /r<br>VFNMSUB213PS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.             |
| VEX.NDS.256.66.0F38.0 BE /r<br>VFNMSUB231PS ymm1, ymm2,<br>ymm3/m256                       | RVM       | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.             |
| EVEX.NDS.128.66.0F38.W0 9E /r<br>VFNMSUB132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.    |
| EVEX.NDS.128.66.0F38.W0 AE /r<br>VFNMSUB213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.    |
| EVEX.NDS.128.66.0F38.W0 BE /r<br>VFNMSUB231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1. |
| EVEX.NDS.256.66.0F38.W0 9E /r<br>VFNMSUB132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.    |
| EVEX.NDS.256.66.0F38.W0 AE /r<br>VFNMSUB213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.    |
| EVEX.NDS.256.66.0F38.W0 BE /r<br>VFNMSUB231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1. |
| EVEX.NDS.512.66.0F38.W0 9E /r<br>VFNMSUB132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.    |
| EVEX.NDS.512.66.0F38.W0 AE /r<br>VFNMSUB213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.    |
| EVEX.NDS.512.66.0F38.W0 BE /r<br>VFNMSUB231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1. |

## Instruction Operand Encoding

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

## Description

**VFNMSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**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.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-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[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```



**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*

```

```

THEN
  IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(-SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i]
    ELSE
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(-SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i]
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFNMSUBxxxPS __m512 __mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 __mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 __mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 __mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNSUB132SD/VFNSUB213SD/VFNSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| VEX.DDS.LIG.66.0F38.W1 9F /r<br>VFNSUB132SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 AF /r<br>VFNSUB213SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W1 BF /r<br>VFNSUB231SD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | FMA                      | Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 9F /r<br>VFNSUB132SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 AF /r<br>VFNSUB213SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W1 BF /r<br>VFNSUB231SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

**VFNSUB132SD:** 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).

**VFNSUB213SD:** 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).

**VFNSUB231SD:** 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 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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.

## Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

### VFNSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) - SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

### VFNSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) - SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]\*SRC3[63:0]) - DEST[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0] ← RoundFPControl\_MXCSR(- (DEST[63:0]\*SRC3[63:0]) - SRC2[63:0])

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0] ← RoundFPControl\_MXCSR(- (SRC2[63:0]\*DEST[63:0]) - SRC3[63:0])

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0] ← RoundFPControl\_MXCSR(- (SRC2[63:0]\*SRC3[63:0]) - DEST[63:0])

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMSUBxxxSD \_\_m128d \_mm\_fnmsub\_round\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_mask\_fnmsub\_sd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c);

VFNMSUBxxxSD \_\_m128d \_mm\_maskz\_fnmsub\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c);

VFNMSUBxxxSD \_\_m128d \_mm\_mask3\_fnmsub\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k);

VFNMSUBxxxSD \_\_m128d \_mm\_mask\_fnmsub\_round\_sd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_maskz\_fnmsub\_round\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_mask3\_fnmsub\_round\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_fnmsub\_sd (\_\_m128d a, \_\_m128d b, \_\_m128d c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNMSSUB132SS/VFNMSSUB213SS/VFNMSSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|-----------------------------|--------------------------|--|
| VEX.DDS.LIG.66.0F38.W0 9F /r<br>VFNMSSUB132SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                         | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 AF /r<br>VFNMSSUB213SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                         | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 BF /r<br>VFNMSSUB231SS xmm1, xmm2,<br>xmm3/m32              | RVM        | V/V                         | FMA                      | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 9F /r<br>VFNMSSUB132SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                         | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 AF /r<br>VFNMSSUB213SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                         | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 BF /r<br>VFNMSSUB231SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | T1S        | V/V                         | AVX512F                  | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RVM   | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

**VFNMSSUB132SS:** 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).

**VFNMSSUB213SS:** 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).

**VFNMSSUB231SS:** 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 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

#### VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

#### VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);

```



```

FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFNMSSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFNMSSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFNMSSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNMSSUBxxxSS __m128 __m_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSSUBxxxSS __m128 __mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSSUBxxxSS __m128 __mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSSUBxxxSS __m128 __mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSSUBxxxSS __m128 __mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSSUBxxxSS __m128 __mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSSUBxxxSS __m128 __mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSSUBxxxSS __m128 __m_fnmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFPCLASSPD—Tests Types Of a Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|-----------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 66 /r ib<br>VFPCLASSPD k2 {k1},<br>xmm2/m128/m64bcst, imm8 | FV        | V/V                         | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.256.66.0F3A.W1 66 /r ib<br>VFPCLASSPD k2 {k1},<br>ymm2/m256/m64bcst, imm8 | FV        | V/V                         | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.512.66.0F3A.W1 66 /r ib<br>VFPCLASSPD k2 {k1},<br>zmm2/m512/m64bcst, imm8 | FV        | V/V                         | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

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

### Description

The FPCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX\_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-23. The classification test for each category is listed in Table 5-13.

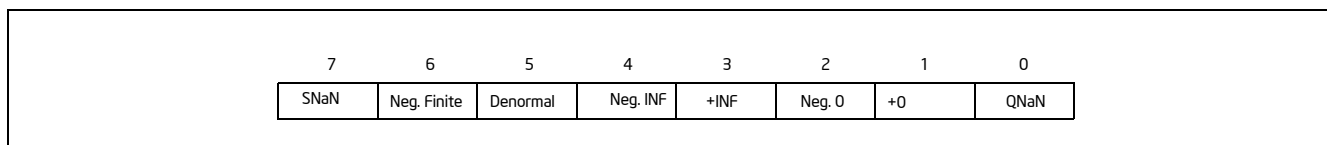


Figure 5-23. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS

Table 5-13. Classifier Operations for VFPCLASSPD/SD/PS/SS

| Bits       | Imm8[0]         | Imm8[1]       | Imm8[2]       | Imm8[3]         | Imm8[4]         | Imm8[5]             | Imm8[6]                    | Imm8[7]         |
|------------|-----------------|---------------|---------------|-----------------|-----------------|---------------------|----------------------------|-----------------|
| Category   | QNaN            | PosZero       | NegZero       | PosINF          | NegINF          | Denormal            | Negative                   | SNaN            |
| Classifier | Checks for QNaN | Checks for +0 | Checks for -0 | Checks for +INF | Checks for -INF | Checks for Denormal | Checks for Negative finite | Checks for SNaN |

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

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

**Operation**

```

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    /* Start checking the source operand for special type */
    NegNum ← tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber ← ExpAllZeros AND MantAllZeros
    SignalingBit ← tsrc[51];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
              ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} /* end of CheckFPClassDP() */

```

**VFPCLASSPD (EVEX Encoded versions)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] ← CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
            FI;
        ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFPCLASSPD __mmask8 __mm512_fpclass_pd_mask( __m512d a, int c);
VFPCLASSPD __mmask8 __mm512_mask_fpclass_pd_mask( __mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 __mm256_fpclass_pd_mask( __m256d a, int c)
VFPCLASSPD __mmask8 __mm256_mask_fpclass_pd_mask( __mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 __mm_fpclass_pd_mask( __m128d a, int c)

```

VFPCLASSPD \_\_mmask8 \_mm\_mask\_fpclass\_pd\_mask( \_\_mmask8 m, \_\_m128d a, int c)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4

#UD                    If EVEX.vvvv != 1111B.

## VFPCLASSPS—Tests Types Of a Packed Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 66 /r ib<br>VFPCLASSPS k2 {k1},<br>xmm2/m128/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.256.66.0F3A.W0 66 /r ib<br>VFPCLASSPS k2 {k1},<br>ymm2/m256/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.512.66.0F3A.W0 66 /r ib<br>VFPCLASSPS k2 {k1},<br>zmm2/m512/m32bcst, imm8 | FV        | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

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

### Description

The FPCLASSPS instruction checks the packed single-precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX\_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-23. The classification test for each category is listed in Table 5-13.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

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

### Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){
```

```
    /* Start checking the source operand for special type */
```

```
    NegNum ← tsrc[31];
```

```
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
```

```
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
```

```
    IF (ExpAllZeros AND MXCSR.DAZ) Then
```

```
        MantAllZeros ← 1;
```

```
    ELSIF (tsrc[22:0]=0h) Then
```

```
        MantAllZeros ← 1;
```

```
    FI;
```

```
    ZeroNumber= ExpAllZeros AND MantAllZeros
```

```
    SignalingBit= tsrc[22];
```

```
    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit) ; // sNaN
```

```
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
```

```
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```

```

Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros;; // -0
PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros;; // +Inf
NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros;; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros);; // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber);; // -finite

bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
           ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;
} /* end of CheckSPClassSP() */

```

**VFPCLASSPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[jj] OR \*no writemask\*

THEN

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

THEN

DEST[jj] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);

ELSE

DEST[jj] ← CheckFPClassDP(SRC1[j+31:i], imm8[7:0]);

FI;

ELSE DEST[jj] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFPCLASSPS \_\_mmask16 \_\_mm512\_fpclass\_ps\_mask( \_\_m512 a, int c);

VFPCLASSPS \_\_mmask16 \_\_mm512\_mask\_fpclass\_ps\_mask( \_\_mmask16 m, \_\_m512 a, int c)

VFPCLASSPS \_\_mmask8 \_\_mm256\_fpclass\_ps\_mask( \_\_m256 a, int c)

VFPCLASSPS \_\_mmask8 \_\_mm256\_mask\_fpclass\_ps\_mask( \_\_mmask8 m, \_\_m256 a, int c)

VFPCLASSPS \_\_mmask8 \_\_mm\_fpclass\_ps\_mask( \_\_m128 a, int c)

VFPCLASSPS \_\_mmask8 \_\_mm\_mask\_fpclass\_ps\_mask( \_\_mmask8 m, \_\_m128 a, int c)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4

#UD If EVEX.vvvv != 1111B.

## VFPCLASSSD—Tests Types Of a Scalar Float64 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LIG.66.0F3A.W1 67 /r ib<br>VFPCLASSSD k2 {k1},<br>xmm2/m64, imm8 | T1S       | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

The FPCLASSSD instruction checks the low double precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-23. The classification test for each category is listed in Table 5-13.

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

### Operation

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

NegNum  $\leftarrow$  tsrc[63];

IF (tsrc[62:52]=07FFh) Then ExpAllOnes  $\leftarrow$  1; FI;

IF (tsrc[62:52]=0h) Then ExpAllZeros  $\leftarrow$  1;

IF (ExpAllZeros AND MXCSR.DAZ) Then

    MantAllZeros  $\leftarrow$  1;

ELSIF (tsrc[51:0]=0h) Then

    MantAllZeros  $\leftarrow$  1;

FI;

ZeroNumber  $\leftarrow$  ExpAllZeros AND MantAllZeros

SignalingBit  $\leftarrow$  tsrc[51];

sNaN\_res  $\leftarrow$  ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN

qNaN\_res  $\leftarrow$  ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN

Pzero\_res  $\leftarrow$  NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0

Nzero\_res  $\leftarrow$  NegNum AND ExpAllZeros AND MantAllZeros; // -0

Plnf\_res  $\leftarrow$  NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf

Nlnf\_res  $\leftarrow$  NegNum AND ExpAllOnes AND MantAllZeros; // -Inf

Denorm\_res  $\leftarrow$  ExpAllZeros AND NOT(MantAllZeros); // denorm

FinNeg\_res  $\leftarrow$  NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN\_res ) OR ( imm8[1] AND Pzero\_res ) OR

    ( imm8[2] AND Nzero\_res ) OR ( imm8[3] AND Plnf\_res ) OR

    ( imm8[4] AND Nlnf\_res ) OR ( imm8[5] AND Denorm\_res ) OR

    ( imm8[6] AND FinNeg\_res ) OR ( imm8[7] AND sNaN\_res );

Return bResult;

} /\* end of CheckFPClassDP() \*/

**VFPCLASSSD (EVEX encoded version)**

```
IF k1[0] OR *no writemask*  
  THEN DEST[0] ←  
    CheckFPClassDP(SRC1[63:0], imm8[7:0])  
  ELSE DEST[0] ← 0 ; zeroing-masking only  
FI;  
DEST[MAX_KL-1:1] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VFPCLASSSD __mmask8 _mm_fpclass_sd_mask( __m128d a, int c)  
VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask( __mmask8 m, __m128d a, int c)
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.



## VFPCLASSSS—Tests Types Of a Scalar Float32 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEEX.LIG.66.0F3A.W0 67 /r<br>VFPCLASSSS k2 {k1},<br>xmm2/m32, imm8 | T1S       | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

The FPCLASSSS instruction checks the low single-precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-23. The classification test for each category is listed in Table 5-13.

EVEEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0]){

```

    /* Start checking the source operand for special type */
    NegNum ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit) ; // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros;; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros;; // -0
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros;; // +Inf
    Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros;; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
              ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;

```

```
} /* end of CheckSPClassSP() */
```

**VFPCLASSSS (EVEX encoded version)**

```
IF k1[0] OR *no writemask*  
  THEN DEST[0] ←  
    CheckFPClassSP(SRC1[31:0], imm8[7:0])  
  ELSE DEST[0] ← 0 ; zeroing-masking only  
FI;  
DEST[MAX_KL-1:1] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VFPCLASSSS __mmask8 _mm_fpclass_ss_mask( __m128 a, int c)  
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask( __mmask8 m, __m128 a, int c)
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.

## VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 90 /vsib<br>VPGATHERDD xmm1 {k1}, vm32x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.256.66.0F38.W0 90 /vsib<br>VPGATHERDD ymm1 {k1}, vm32y | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.512.66.0F38.W0 90 /vsib<br>VPGATHERDD zmm1 {k1}, vm32z | T1S       | V/V                          | AVX512F                  | Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.128.66.0F38.W1 90 /vsib<br>VPGATHERDQ xmm1 {k1}, vm32x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.256.66.0F38.W1 90 /vsib<br>VPGATHERDQ ymm1 {k1}, vm32x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.512.66.0F38.W1 90 /vsib<br>VPGATHERDQ zmm1 {k1}, vm32y | T1S       | V/V                          | AVX512F                  | Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---|-----------|-----------|
| T1S   | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        |

### Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into vector `zmm1`. The elements are specified via the `VSIB` (i.e., the index register is a `zmm`, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (`zmm1`) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

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 register (`k1`) 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 element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements 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.

Note that:

- 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 `zmm` 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.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same  $\text{disp8} * N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

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

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPGATHERDD (EVEX encoded version)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j]

    THEN DEST[i+31:i] ← MEM[BASE\_ADDR +  
      SignExtend(VINDEX[i+31:i]) \* SCALE + DISP], 1)

    k1[j] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\*       ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

#### VPGATHERDQ (EVEX encoded version)

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j]

    THEN DEST[i+63:i] ←  
      MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP]

    k1[j] ← 0

    ELSE \*DEST[i+63:i] ← remains unchanged\*       ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPGATHERDD __m512i __mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i __mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i __mm256_mask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i __mm_mask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i __mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i __mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i __mm256_mask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i __mm_mask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 91 /vsib<br>VPGATHERQD xmm1 {k1}, vm64x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.256.66.0F38.W0 91 /vsib<br>VPGATHERQD xmm1 {k1}, vm64y | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.512.66.0F38.W0 91 /vsib<br>VPGATHERQD ymm1 {k1}, vm64z | T1S       | V/V                          | AVX512F                  | Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.128.66.0F38.W1 91 /vsib<br>VPGATHERQQ xmm1 {k1}, vm64x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.256.66.0F38.W1 91 /vsib<br>VPGATHERQQ ymm1 {k1}, vm64y | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.512.66.0F38.W1 91 /vsib<br>VPGATHERQQ zmm1 {k1}, vm64z | T1S       | V/V                          | AVX512F                  | Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---|-----------|-----------|
| T1S   | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        |

### Description

A set of 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the `VSIB` (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

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 register (`k1`) 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 element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements 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.

Note that:

- 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 `zmm` 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.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same  $\text{disp}8*N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

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

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPGATHERQD (EVEX encoded version)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  k ← j \* 64

  IF k1[j]

    THEN DEST[i+31:i] ← MEM[BASE\_ADDR + (VINDEX[k+63:k]) \* SCALE + DISP], 1)

    k1[j] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\* ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL/2] ← 0

#### VPGATHERQQ (EVEX encoded version)

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j]

    THEN DEST[i+63:i] ←

      MEM[BASE\_ADDR + (VINDEX[i+63:i]) \* SCALE + DISP]

    k1[j] ← 0

    ELSE \*DEST[i+63:i] ← remains unchanged\* ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```
VPGATHERQD __m256i __mm512_i64gather_epi32(__m512i vdx, void * base, int scale);  
VPGATHERQD __m256i __mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);  
VPGATHERQD __m128i __mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);  
VPGATHERQD __m128i __mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);  
VPGATHERQQ __m512i __mm512_i64gather_epi64(__m512i vdx, void * base, int scale);  
VPGATHERQQ __m512i __mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);  
VPGATHERQQ __m256i __mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);  
VPGATHERQQ __m128i __mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E12.



## VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 92 /vsib<br>VGATHERDPS xmm1 {k1}, vm32x    | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EVEX.256.66.0F38.W0 92 /vsib<br>VGATHERDPS ymm1 {k1}, vm32y    | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EVEX.512.66.0F38.W0 92 /vsib<br>VGATHERDPS zmm1 {k1}, vm32z    | T1S       | V/V                          | AVX512F                  | Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EVEX.128.66.0F38.W1 92 /vsib<br>VGATHERDPD xmm1 {k1},<br>vm32x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.            |
| EVEX.256.66.0F38.W1 92 /vsib<br>VGATHERDPD ymm1 {k1},<br>vm32x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.            |
| EVEX.512.66.0F38.W1 92 /vsib<br>VGATHERDPD zmm1 {k1}, vm32y    | T1S       | V/V                          | AVX512F                  | Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.            |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---|-----------|-----------|
| T1S   | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        |

### Description

A set of single-precision/double-precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

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 right most one with its mask bit set). When this happens, the destination register and the mask register (k1) 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 element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements 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.

Note that:

- 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 zmm 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.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

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.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VGATHERDPS (EVEX encoded version)

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

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 32$

  IF  $k1[j]$

    THEN  $\text{DEST}[i+31:i] \leftarrow$

$\text{MEM}[\text{BASE\_ADDR} +$

$\text{SignExtend}(\text{VINDEX}[i+31:i]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

    ELSE  $*\text{DEST}[i+31:i] \leftarrow$  remains unchanged\*

  FI;

ENDFOR

$k1[\text{MAX\_KL}-1:\text{KL}] \leftarrow 0$

$\text{DEST}[\text{MAX\_VL}-1:\text{VL}] \leftarrow 0$

#### VGATHERDPD (EVEX encoded version)

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

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 64$

$k \leftarrow j * 32$

  IF  $k1[j]$

    THEN  $\text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE\_ADDR} +$

$\text{SignExtend}(\text{VINDEX}[k+31:k]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

    ELSE  $*\text{DEST}[i+63:i] \leftarrow$  remains unchanged\*

  FI;

ENDFOR

$k1[\text{MAX\_KL}-1:\text{KL}] \leftarrow 0$

$\text{DEST}[\text{MAX\_VL}-1:\text{VL}] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGATHERDPD __m512d __mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d __mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d __mm256_mask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d __mm_mask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 __mm256_mask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 __mm_mask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 93 /vsib<br>VGATHERQPS xmm1 {k1}, vm64x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EVEX.256.66.0F38.W0 93 /vsib<br>VGATHERQPS xmm1 {k1}, vm64y | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EVEX.512.66.0F38.W0 93 /vsib<br>VGATHERQPS ymm1 {k1}, vm64z | T1S       | V/V                          | AVX512F                  | Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EVEX.128.66.0F38.W1 93 /vsib<br>VGATHERQPD xmm1 {k1}, vm64x | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.            |
| EVEX.256.66.0F38.W1 93 /vsib<br>VGATHERQPD ymm1 {k1}, vm64y | T1S       | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.            |
| EVEX.512.66.0F38.W1 93 /vsib<br>VGATHERQPD zmm1 {k1}, vm64z | T1S       | V/V                          | AVX512F                  | Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.            |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---|-----------|-----------|
| T1S   | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        |

### Description

A set of 8 single-precision/double-precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

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 register (k1) 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 element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements 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.

Note that:

- 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 zmm 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.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules.  $N$  is considered to be the size of a single vector element. 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.

The instruction will #UD fault if the destination vector `zmm1` is the same as index vector `VINDEX`. The instruction will #UD fault if the `k0` mask register is specified.

### Operation

`BASE_ADDR` stands for the memory operand base address (a GPR); may not exist

`VINDEX` stands for the memory operand vector of indices (a ZMM register)

`SCALE` stands for the memory operand scalar (1, 2, 4 or 8)

`DISP` is the optional 1, 2 or 4 byte displacement

### VGATHERQPS (EVEX encoded version)

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

$k \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN  $\text{DEST}[i+31:i] \leftarrow$

$\text{MEM}[\text{BASE\_ADDR} + (\text{VINDEX}[k+63:k]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

ELSE \* $\text{DEST}[i+31:i] \leftarrow$  remains unchanged\*

FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

$\text{DEST}[\text{MAX\_VL}-1:VL/2] \leftarrow 0$

### VGATHERQPD (EVEX encoded version)

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

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN  $\text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE\_ADDR} + (\text{VINDEX}[i+63:i]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

ELSE \* $\text{DEST}[i+63:i] \leftarrow$  remains unchanged\*

FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

$\text{DEST}[\text{MAX\_VL}-1:VL] \leftarrow 0$

### Intel C/C++ Compiler Intrinsic Equivalent

```
VGATHERQPD __m512d __mm512_i64gather_pd(__m512i vdx, void * base, int scale);  
VGATHERQPD __m512d __mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);  
VGATHERQPD __m256d __mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);  
VGATHERQPD __m128d __mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);  
VGATHERQPS __m256 __mm512_i64gather_ps(__m512i vdx, void * base, int scale);  
VGATHERQPS __m256 __mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);  
VGATHERQPS __m128 __mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);  
VGATHERQPS __m128 __mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E12.

## VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 42 /r<br>VGETEXPPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.           |
| EVEX.256.66.0F38.W1 42 /r<br>VGETEXPPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.           |
| EVEX.512.66.0F38.W1 42 /r<br>VGETEXPPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | FV        | V/V                          | AVX512F                  | Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1. |

### Instruction Operand Encoding

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

### Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-14.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

**Table 5-14. VGETEXPPD/SD Special Cases**

| Input Operand      | Result                         | Comments      |
|--------------------|--------------------------------|---------------|
| src1 = NaN         | QNaN(src1)                     | No Exceptions |
| $0 <  src1  < INF$ | $\text{floor}(\log_2( src1 ))$ |               |
| $ src1  = +INF$    | +INF                           |               |
| $ src1  = 0$       | -INF                           |               |

### Operation

NormalizeExpTinyDPFP(SRC[63:0])

```
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
```

```

Dst.exp ← 1;
Dst.fraction ← SRC[51:0];
WHILE(Src.lbit = 0)
{
    Src.lbit ← Dst.fraction[51];    // Get the fraction MSB
    Dst.fraction ← Dst.fraction << 1;    // One bit shift left
    Dst.exp--;    // Decrement the exponent
}
Dst.fraction ← 0;    // zero out fraction bits
Dst.sign ← 1;    // Return negative sign
TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (TMP[63:0]);
}

```

ConvertExpDPFP(SRC[63:0])

```

{
    Src.sign ← 0;    // Zero out sign bit
    Src.exp ← SRC[62:52];
    Src.fraction ← SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE    // check if denormal operand (notice that MXCSR.DAZ = 0)
{
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
        TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]);    // Get Normalized Exponent
        Set #DE
    }
    ELSE    // exponent value is correct
    {
        Dst.fraction ← 0;    // zero out fraction bits
        TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
    }
    TMP ← SAR(TMP, 52);    // Shift Arithmetic Right
    TMP ← TMP - 1023;    // Subtract Bias
    Return CvtI2D(TMP);    // Convert INT to Double-Precision FP number
}
}

```

#### VGETEXPPD (EVEX encoded versions)

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*



```

THEN
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN
      DEST[j+63:i] ←
      ConvertExpDPFP(SRC[63:0])
    ELSE
      DEST[j+63:i] ←
      ConvertExpDPFP(SRC[j+63:i])
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[j+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd(__mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd(__mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd(__mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd(__mmask8 k, __m128d a);

```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 42 /r<br>VGETEXPPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register. |
| EVEX.256.66.0F38.W0 42 /r<br>VGETEXPPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register. |
| EVEX.512.66.0F38.W0 42 /r<br>VGETEXPPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae} | FV        | V/V                          | AVX512F                  | Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register. |

### Instruction Operand Encoding

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

### Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Table 5-15. VGETEXPPS/SS Special Cases

| Input Operand    | Result                           | Comments      |
|------------------|----------------------------------|---------------|
| src1 = NaN       | QNaN(src1)                       | No Exceptions |
| 0 <  src1  < INF | floor(log <sub>2</sub> ( src1 )) |               |
| src1  = +INF     | +INF                             |               |
| src1  = 0        | -INF                             |               |

Figure 5-24 illustrates the VGETEXPPS functionality on input values with normalized representation.

|                                 | 31 | 30  | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21       | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------------------|----|-----|----|----|----|----|----|----|----|----|----------|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
|                                 | s  | exp |    |    |    |    |    |    |    |    | Fraction |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
| Src = 2 <sup>M</sup>            | 0  | 1   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SAR Src, 23 = 080h              | 0  | 0   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| -Bias                           | 1  | 1   | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1        | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Tmp - Bias = 1                  | 0  | 0   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Out_P12PS(01h) = 2 <sup>0</sup> | 0  | 0   | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 5-24. VGETEXPPS Functionality On Normal Input values

**Operation**

```
NormalizeExpTinySPFP(SRC[31:0])
```

```
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[22:0];
    WHILE(Src.jbit = 0)
    {
        Src.jbit ← Dst.fraction[22]; // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1; // One bit shift left
        Dst.exp--; // Decrement the exponent
    }
    Dst.fraction ← 0; // zero out fraction bits
    Dst.sign ← 1; // Return negative sign
    TMP[31:0] ← MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
    Return (TMP[31:0]);
}
```

```
ConvertExpSPFP(SRC[31:0])
```

```
{
    Src.sign ← 0; // Zero out sign bit
    Src.exp ← SRC[30:23];
    Src.fraction ← SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF (SRC = SNAN) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
```

```

        TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]);    // Get Normalized Exponent
        Set #DE
    }
    ELSE      // exponent value is correct
    {
        Dst.fraction ← 0;          // zero out fraction bits
        TMP[31:0] ← (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
    }
    TMP ← SAR(TMP, 23);           // Shift Arithmetic Right
    TMP ← TMP - 127;             // Subtract Bias
    Return CvtI2D(TMP);         // Convert INT to Single-Precision FP number
}
}

```

**VGETEXPPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

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

THEN

DEST[i+31:i] ←

ConvertExpSPFP(SRC[31:0])

ELSE

DEST[i+31:i] ←

ConvertExpSPFP(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPPS \_\_m512 \_\_mm512\_getexp\_ps(\_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_mask\_getexp\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_maskz\_getexp\_ps(\_\_mmask16 k, \_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_getexp\_round\_ps(\_\_m512 a, int sae);

VGETEXPPS \_\_m512 \_\_mm512\_mask\_getexp\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, int sae);

VGETEXPPS \_\_m512 \_\_mm512\_maskz\_getexp\_round\_ps(\_\_mmask16 k, \_\_m512 a, int sae);

VGETEXPPS \_\_m256 \_\_mm256\_getexp\_ps(\_\_m256 a);

VGETEXPPS \_\_m256 \_\_mm256\_mask\_getexp\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VGETEXPPS \_\_m256 \_\_mm256\_maskz\_getexp\_ps(\_\_mmask8 k, \_\_m256 a);

VGETEXPPS \_\_m128 \_\_mm\_getexp\_ps(\_\_m128 a);

VGETEXPPS \_\_m128 \_\_mm\_mask\_getexp\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VGETEXPPS \_\_m128 \_\_mm\_maskz\_getexp\_ps(\_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F38.W1 43 /r<br>VGETEXPSD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae} | T1S       | V/V                         | AVX512F                  | Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-14.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

### Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

#### VGETEXPSD (EVEX encoded version)

```

IF k1[0] OR *no writemask*
  THEN DEST[63:0] ←
    ConvertExpDPFP(SRC2[63:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[63:0] ← 0
  FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VGETEXPSD __m128d _mm_getexp_sd( __m128d a, __m128d b);  
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);  
VGETEXPSD __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);  
VGETEXPSD __m128d _mm_getexp_round_sd( __m128d a, __m128d b, int sae);  
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);  
VGETEXPSD __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.OF38.W0 43 /r<br>VGETEXPSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae} | T1S       | V/V                          | AVX512F                  | Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

### Operation

```
// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPSS
```

```
// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPSS
```

#### VGETEXPSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
  THEN DEST[31:0] ←
    ConvertExpDPFP(SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[31:0] ← 0
    FI
  FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```



**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPSS \_\_m128\_mm\_getexp\_ss( \_\_m128 a, \_\_m128 b);  
VGETEXPSS \_\_m128\_mm\_mask\_getexp\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
VGETEXPSS \_\_m128\_mm\_maskz\_getexp\_ss(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
VGETEXPSS \_\_m128\_mm\_getexp\_round\_ss( \_\_m128 a, \_\_m128 b, int sae);  
VGETEXPSS \_\_m128\_mm\_mask\_getexp\_round\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int sae);  
VGETEXPSS \_\_m128\_mm\_maskz\_getexp\_round\_ss( \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W1 26 /r ib<br>VGETMANTPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512F      | Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask. |
| EVEX.256.66.0F3A.W1 26 /r ib<br>VGETMANTPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512F      | Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask. |
| EVEX.512.66.0F3A.W1 26 /r ib<br>VGETMANTPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae},<br>imm8 | FV        | V/V                          | AVX512F                  | Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FVI   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-25. The converted results are written to the destination operand (the first operand) using writemask *k1*. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

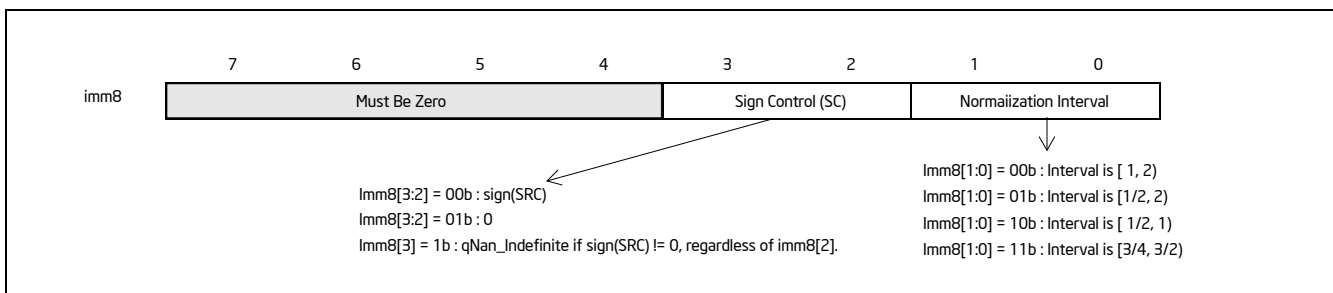


Figure 5-25. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value *x*, The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If  $interv \neq 0$  then  $k = -1$ , otherwise  $k = 0$ . The encoded value of  $imm8[1:0]$  and sign control are shown in Figure 5-25.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent  $k$  (adding bias) and a mantissa normalized to the range specified by  $interv$ .

The `GetMant()` function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register  $k1$  are computed and stored into the destination. Elements in  $zmm1$  with the corresponding bit clear in  $k1$  retain their previous values.

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

**Table 5-16. GetMant() Special Float Values Behavior**

| Input     | Result   | Exceptions / Comments                            |
|-----------|--|--|
| NaN       | QNaN(SRC)  | Ignore <i>interv</i><br>If (SRC = SNaN) then #IE |
| $+\infty$ | 1.0  | Ignore <i>interv</i>                             |
| +0        | 1.0  | Ignore <i>interv</i>                             |
| -0        | IF (SC[0]) THEN +1.0<br>ELSE -1.0  | Ignore <i>interv</i>                             |
| $-\infty$ | IF (SC[1]) THEN {QNaN_Indefinite}<br>ELSE {<br>IF (SC[0]) THEN +1.0<br>ELSE -1.0 | Ignore <i>interv</i><br>If (SC[1]) then #IE      |
| negative  | SC[1] ? QNaN_Indefinite : Getmant(SRC)   | If (SC[1]) then #IE                              |

### Operation

```
GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];           // Get sign bit
    Dst.exp ← SRC[62:52]; // Get original exponent value
    Dst.fraction ← SRC[51:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 03FFh; // Override exponent with BIAS
        Return ((Dst.sign << 63) | (Dst.exp << 52) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
}
```

```

// Checking for denormal operands
IF (DenormOperand)
{
  IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
  ELSE
  {
    // Jbit is the hidden integral bit. Zero in case of denormal operand.
    Src.Jbit ← 0; // Zero Src Jbit
    Dst.exp ← 03FFh; // Override exponent with BIAS
    WHILE (Src.Jbit = 0) { // normalize mantissa
      Src.Jbit ← Dst.fraction[51]; // Get the fraction MSB
      Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
      Dst.exp-- ; // Adjust the exponent
    }
    SET #DE; // Set DE bit
  }
}
// At this point, Dst.fraction is normalized.
// Checking for exponent response
Unbiased.exp ← Dst.exp - 03FFh; // subtract the bias from exponent
IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];
CASE (interv[1:0])
  00: Dst.exp ← 03FFh; // This is the bias
  01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh; // either bias-1, or bias
  10: Dst.exp ← 03FEh; // bias-1
  11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh; // either bias-1, or bias
ESCA
// At this point Dst.exp has the correct result. Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (DEST);
}

```

```

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

```

**VGETMANTPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

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

        THEN

          DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[63:0], sc, interv)

        ELSE

          DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[j+63:i], sc, interv)

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

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

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETMANTPD __m512d __mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d __mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d __mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d __mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d __mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d __mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

```

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

**VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector**

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 26 /r ib<br>VGETMANTPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512F      | Get normalized mantissa from float32 vector<br>xmm2/m128/m32bcst and store the result in xmm1, using<br>imm8 for sign control and mantissa interval normalization,<br>under writemask. |
| EVEX.256.66.0F3A.W0 26 /r ib<br>VGETMANTPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512F      | Get normalized mantissa from float32 vector<br>ymm2/m256/m32bcst and store the result in ymm1, using<br>imm8 for sign control and mantissa interval normalization,<br>under writemask. |
| EVEX.512.66.0F3A.W0 26 /r ib<br>VGETMANTPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae},<br>imm8 | FV        | V/V                          | AVX512F                  | Get normalized mantissa from float32 vector<br>zmm2/m512/m32bcst and store the result in zmm1, using<br>imm8 for sign control and mantissa interval normalization,<br>under writemask. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FVI   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

**Description**

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-25. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input SP FP value  $x$ , The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent  $k$  depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then  $k = -1$ , otherwise  $K = 0$ . The encoded value of imm8[1:0] and sign control are shown in Figure 5-25.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent  $k$  (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

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

**Operation**

```

GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];           // Get sign bit
    Dst.exp ← SRC[30:23]; // Get original exponent value
    Dst.fraction ← SRC[22:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 07Fh; // Override exponent with BIAS
        Return ((Dst.sign << 31) | (Dst.exp << 23) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {
        IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
        ELSE
        {
            // Jbit is the hidden integral bit. Zero in case of denormal operand.
            Src.Jbit ← 0; // Zero Src Jbit
            Dst.exp ← 07Fh; // Override exponent with BIAS
            WHILE (Src.Jbit = 0) { // normalize mantissa
                Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
                Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
                Dst.exp--; // Adjust the exponent
            }
            SET #DE; // Set DE bit
        }
    }
    // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp - 07Fh; // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[22];
    CASE (interv[1:0])
        00: Dst.exp ← 07Fh; // This is the bias
        01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh; // either bias-1, or bias
        10: Dst.exp ← 07Eh; // bias-1
        11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh; // either bias-1, or bias
    ESCA

    // Form the final destination
    DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
}

```

```

    Return (DEST);
}

```

```

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

```

**VGETMANTPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

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

THEN

DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[31:0], sc, interv)

ELSE

DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[i+31:i], sc, interv)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTPS \_\_m512 \_\_mm512\_getmant\_ps( \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_mask\_getmant\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_maskz\_getmant\_ps(\_\_mmask16 k, \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_getmant\_round\_ps( \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m512 \_\_mm512\_mask\_getmant\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m512 \_\_mm512\_maskz\_getmant\_round\_ps(\_\_mmask16 k, \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m256 \_\_mm256\_getmant\_ps( \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m256 \_\_mm256\_mask\_getmant\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m256 \_\_mm256\_maskz\_getmant\_ps( \_\_mmask8 k, \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_getmant\_ps( \_\_m128 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_mask\_getmant\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_maskz\_getmant\_ps( \_\_mmask8 k, \_\_m128 a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.



## VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F3A.W1 27 /r ib<br>VGETMANTSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{sae}, imm8 | T1S       | V/V                          | AVX512F                  | Extract the normalized mantissa of the low float64 element in xmm3/m64 using <i>imm8</i> for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-25. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If *interv* != 0 then *k* = -1, otherwise *k* = 0. The encoded value of *imm8*[1:0] and sign control are shown in Figure 5-25.

The converted DP FP result is encoded according to the sign control, the unbiased exponent *k* (adding bias) and a mantissa normalized to the range specified by *interv*.

The *GetMant()* function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

### Operation

// *GetNormalizeMantissaDP*(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of *VGETMANTPD*

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

### VGETMANTSD (EVEX encoded version)

IF k1[0] OR \*no writemask\*

THEN DEST[63:0] ←

*GetNormalizedMantissaDP*(SRC2[63:0], *sc*, *interv*)

ELSE

    IF \*merging-masking\* ; merging-masking

```

        THEN *DEST[63:0] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[63:0] ← 0
    FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

```

#### SIMD Floating-Point Exceptions

Denormal, Invalid

#### Other Exceptions

See Exceptions Type E3.

## VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F3A.W0 27 /r ib<br>VGETMANTSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae}, imm8 | T1S       | V/V                          | AVX512F                  | Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-25. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-25.

The converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

### Operation

// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

#### VGETMANTSS (EVEX encoded version)

IF k1[0] OR \*no writemask\*

THEN DEST[31:0] ←

GetNormalizedMantissaSP(SRC2[31:0], sc, interv)

ELSE

IF \*merging-masking\* ; merging-masking

```

        THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[31:0] ← 0
    FI
FI;
DEST[127:32] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

```

#### SIMD Floating-Point Exceptions

Denormal, Invalid

#### Other Exceptions

See Exceptions Type E3.

## VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| VEX.NDS.256.66.0F3A.W0 18 /r ib<br>VINSERTF128 ymm1, ymm2,<br>xmm3/m128, imm8           | RVM1       | V/V                          | AVX                      | Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.                                     |
| EVEX.NDS.256.66.0F3A.W0 18 /r ib<br>VINSERTF32X4 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | T4         | V/V                          | AVX512VL<br>AVX512F      | Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F3A.W0 18 /r ib<br>VINSERTF32X4 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | T4         | V/V                          | AVX512F                  | Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.NDS.256.66.0F3A.W1 18 /r ib<br>VINSERTF64X2 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | T2         | V/V                          | AVX512VL<br>AVX512DQ     | Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F3A.W1 18 /r ib<br>VINSERTF64X2 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | T2         | V/V                          | AVX512DQ                 | Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.NDS.512.66.0F3A.W0 1A /r ib<br>VINSERTF32X8 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | T8         | V/V                          | AVX512DQ                 | Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.NDS.512.66.0F3A.W1 1A /r ib<br>VINSERTF64X4 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | T4         | V/V                          | AVX512F                  | Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En      | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|------------|---------------|-----------|---------------|-----------|
| RVM1       | ModRM:reg (w) | VEX.vvvv  | ModRM:r/m (r) | Imm8      |
| T2, T4, T8 | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory

location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

### Operation

#### VINSERTF32x4 (EVEX encoded versions)

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

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VINSERTF64x2 (EVEX encoded versions)

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

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

```

IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTF32x8 (EVEX.U1.512 encoded version)**

```

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC2[255:0]
  1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 15
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTF64x4 (EVEX.512 encoded version)**

```

VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC2[255:0]
  1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 7
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTF128 (VEX encoded version)**

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**

VINSERTF32x4 \_\_m512 \_\_mm512\_insertf32x4(\_\_m512 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m512 \_\_mm512\_mask\_insertf32x4(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m512 \_\_mm512\_maskz\_insertf32x4(\_\_mmask16 k, \_\_m512 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m256 \_\_mm256\_insertf32x4(\_\_m256 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m256 \_\_mm256\_mask\_insertf32x4(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m256 \_\_mm256\_maskz\_insertf32x4(\_\_mmask8 k, \_\_m256 a, \_\_m128 b, int imm);

VINSERTF32x8 \_\_m512 \_\_mm512\_insertf32x8(\_\_m512 a, \_\_m256 b, int imm);

VINSERTF32x8 \_\_m512 \_\_mm512\_mask\_insertf32x8(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m256 b, int imm);

VINSERTF32x8 \_\_m512 \_\_mm512\_maskz\_insertf32x8(\_\_mmask16 k, \_\_m512 a, \_\_m256 b, int imm);

VINSERTF64x2 \_\_m512d \_\_mm512\_insertf64x2(\_\_m512d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m512d \_\_mm512\_mask\_insertf64x2(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m512d \_\_mm512\_maskz\_insertf64x2(\_\_mmask8 k, \_\_m512d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m256d \_\_mm256\_insertf64x2(\_\_m256d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m256d \_\_mm256\_mask\_insertf64x2(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m256d \_\_mm256\_maskz\_insertf64x2(\_\_mmask8 k, \_\_m256d a, \_\_m128d b, int imm);

VINSERTF64x4 \_\_m512d \_\_mm512\_insertf64x4(\_\_m512d a, \_\_m256d b, int imm);

VINSERTF64x4 \_\_m512d \_\_mm512\_mask\_insertf64x4(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m256d b, int imm);

VINSERTF64x4 \_\_m512d \_\_mm512\_maskz\_insertf64x4(\_\_mmask8 k, \_\_m512d a, \_\_m256d b, int imm);

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**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.



## VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.NDS.256.66.0F3A.W0 38 /r ib<br>VINSERTI128 ymm1, ymm2,<br>xmm3/m128, imm8           | RVMI       | V/V                          | AVX2                     | Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.  |
| EVEX.NDS.256.66.0F3A.W0 38 /r ib<br>VINSERTI32X4 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | T4         | V/V                          | AVX512VL<br>AVX512F      | Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F3A.W0 38 /r ib<br>VINSERTI32X4 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | T4         | V/V                          | AVX512F                  | Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.NDS.256.66.0F3A.W1 38 /r ib<br>VINSERTI64X2 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | T2         | V/V                          | AVX512VL<br>AVX512DQ     | Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.   |
| EVEX.NDS.512.66.0F3A.W1 38 /r ib<br>VINSERTI64X2 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | T2         | V/V                          | AVX512DQ                 | Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.   |
| EVEX.NDS.512.66.0F3A.W0 3A /r ib<br>VINSERTI32X8 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | T8         | V/V                          | AVX512DQ                 | Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.NDS.512.66.0F3A.W1 3A /r ib<br>VINSERTI64X4 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | T4         | V/V                          | AVX512F                  | Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.   |

### Instruction Operand Encoding

| Op/En      | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|------------|---------------|-----------|---------------|-----------|
| RVMI       | ModRM:reg (w) | VEX.vvvv  | ModRM:r/m (r) | Imm8      |
| T2, T4, T8 | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | Imm8      |

### Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the 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, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

### Operation

#### VINSERTI32x4 (EVEX encoded versions)

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

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VINSERTI64x2 (EVEX encoded versions)

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

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTI32x8 (EVEX.U1.512 encoded version)**

```

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC2[255:0]
  1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 15
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTI64x4 (EVEX.512 encoded version)**

```

VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC2[255:0]
  1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 7
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR

```

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**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**

```

VINSERTI32x4 __m512i _inserti32x4( __m512i a, __m128i b, int imm);
VINSERTI32x4 __m512i _mask_inserti32x4( __m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m512i _maskz_inserti32x4( __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_inserti32x4( __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_mask_inserti32x4( __m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_maskz_inserti32x4( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 __m512i _mm512_inserti32x8( __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_mask_inserti32x8( __m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_maskz_inserti32x8( __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 __m512i _mm512_inserti64x2( __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_mask_inserti64x2( __m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_maskz_inserti64x2( __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_inserti64x2( __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_mask_inserti64x2( __m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_maskz_inserti64x2( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x4 __m512i _mm512_inserti64x4( __m512i a, __m256i b, int imm);
VINSERTI64x4 __m512i _mm512_mask_inserti64x4( __m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 __m512i _mm512_maskz_inserti64x4( __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i _mm256_insertf128_si256( __m256i a, __m128i b, int offset);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

## INSERTPS—Insert Scalar Single-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 3A 21 /r ib<br>INSERTPS xmm1, xmm2/m32, imm8                          | RMI        | V/V                          | SSE4_1                   | Insert a single-precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.  |
| VEX.NDS.128.66.0F3A.WIG 21 /r ib<br>VINSERTPS xmm1, xmm2,<br>xmm3/m32, imm8 | RVMI       | V/V                          | AVX                      | Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8. |
| EVEX.NDS.128.66.0F3A.W0 21 /r ib<br>VINSERTPS xmm1, xmm2,<br>xmm3/m32, imm8 | T1S        | V/V                          | AVX512F                  | Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | Imm8      |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | Imm8      |

### Description

(register source form)

Select a single-precision floating-point element from second source as indicated by Count\_S bits of the immediate operand and destination operand it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

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

### Operation

#### VINSERTPS (VEX.128 and EVEX encoded version)

```
IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
```

```

0: TMP ← SRC2[31:0]
1: TMP ← SRC2[63:32]
2: TMP ← SRC2[95:64]
3: TMP ← SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
0: TMP2[31:0] ← TMP
   TMP2[127:32] ← SRC1[127:32]
1: TMP2[63:32] ← TMP
   TMP2[31:0] ← SRC1[31:0]
   TMP2[127:64] ← SRC1[127:64]
2: TMP2[95:64] ← TMP
   TMP2[63:0] ← SRC1[63:0]
   TMP2[127:96] ← SRC1[127:96]
3: TMP2[127:96] ← TMP
   TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
ELSE DEST[127:96] ← TMP2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**INSERTPS (128-bit Legacy SSE version)**

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
0: TMP ← SRC[31:0]
1: TMP ← SRC[63:32]
2: TMP ← SRC[95:64]
3: TMP ← SRC[127:96]
ESAC;

CASE (COUNT_D) OF
0: TMP2[31:0] ← TMP
   TMP2[127:32] ← DEST[127:32]
1: TMP2[63:32] ← TMP
   TMP2[31:0] ← DEST[31:0]
   TMP2[127:64] ← DEST[127:64]
2: TMP2[95:64] ← TMP
   TMP2[63:0] ← DEST[63:0]
   TMP2[127:96] ← DEST[127:96]
3: TMP2[127:96] ← TMP
   TMP2[95:0] ← DEST[95:0]
ESAC;

```

```

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H

```

```

ELSE DEST[31:0] ←TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ←00000000H
ELSE DEST[63:32] ←TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ←00000000H
ELSE DEST[95:64] ←TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ←00000000H
ELSE DEST[127:96] ←TMP2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

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

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E9NF.

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

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

**Instruction Operand Encoding**

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

**Description**

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

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

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

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

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



VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

### Operation

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

### VMAXPD (EVEX encoded versions)

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

```
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[63:0])
        ELSE
          DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0 ; zeroing-masking
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

### VMAXPD (VEX.256 encoded version)

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

### VMAXPD (VEX.128 encoded version)

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

**MAXPD (128-bit Legacy SSE version)**

DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])

DEST[127:64] ← MAX(DEST[127:64], SRC[127:64])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMAXPD \_\_m512d \_mm512\_max\_pd( \_\_m512d a, \_\_m512d b);

VMAXPD \_\_m512d \_mm512\_mask\_max\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b,);

VMAXPD \_\_m512d \_mm512\_maskz\_max\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VMAXPD \_\_m512d \_mm512\_max\_round\_pd( \_\_m512d a, \_\_m512d b, int);

VMAXPD \_\_m512d \_mm512\_mask\_max\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMAXPD \_\_m512d \_mm512\_maskz\_max\_round\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMAXPD \_\_m256d \_mm256\_mask\_max\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VMAXPD \_\_m256d \_mm256\_maskz\_max\_pd( \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VMAXPD \_\_m128d \_mm\_mask\_max\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VMAXPD \_\_m128d \_mm\_maskz\_max\_pd( \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VMAXPD \_\_m256d \_mm256\_max\_pd( \_\_m256d a, \_\_m256d b);

(V)VMAXPD \_\_m128d \_mm\_max\_pd( \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E2.

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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| OF 5F /r<br>MAXPS xmm1, xmm2/m128  | RM        | V/V                          | SSE                      | Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.  |
| VEX.NDS.128.OF.WIG 5F /r<br>VMAXPS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | AVX                      | Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.  |
| VEX.NDS.256.OF.WIG 5F /r<br>VMAXPS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | AVX                      | Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.  |
| EVEX.NDS.128.OF.W0 5F /r<br>VMAXPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1. |
| EVEX.NDS.256.OF.W0 5F /r<br>VMAXPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1. |
| EVEX.NDS.512.OF.W0 5F /r<br>VMAXPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{sae} | FV        | V/V                          | AVX512F                  | Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

### Operation

MAX(SRC1, SRC2)

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

### VMAXPS (EVEX encoded versions)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

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

        THEN

          DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])

        ELSE

          DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VMAXPS (VEX.256 encoded version)

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])

DEST[159:128] ← MAX(SRC1[159:128], SRC2[159:128])

DEST[191:160] ← MAX(SRC1[191:160], SRC2[191:160])

DEST[223:192] ← MAX(SRC1[223:192], SRC2[223:192])

DEST[255:224] ← MAX(SRC1[255:224], SRC2[255:224])

DEST[MAX\_VL-1:256] ← 0

**VMAXPS (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96])$   
 $\text{DEST}[\text{MAX\_VL}-1:128] \leftarrow 0$

**MAXPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{DEST}[31:0], \text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{DEST}[63:32], \text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{DEST}[95:64], \text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{DEST}[127:96], \text{SRC}[127:96])$   
 $\text{DEST}[\text{MAX\_VL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VMAXPS __m512 __mm512_max_ps( __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_maskz_max_ps( __mmask16 k, __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_max_round_ps( __m512 a, __m512 b, int);`  
`VMAXPS __m512 __mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VMAXPS __m512 __mm512_maskz_max_round_ps( __mmask16 k, __m512 a, __m512 b, int);`  
`VMAXPS __m256 __mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VMAXPS __m256 __mm256_maskz_max_ps( __mmask8 k, __m256 a, __m256 b);`  
`VMAXPS __m128 __mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VMAXPS __m128 __mm_maskz_max_ps( __mmask8 k, __m128 a, __m128 b);`  
`VMAXPS __m256 __mm256_max_ps( __m256 a, __m256 b);`  
`MAXPS __m128 __mm_max_ps( __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E2.

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

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| F2 0F 5F /r<br>MAXSD xmm1, xmm2/m64  | RM         | V/V                          | SSE2                     | Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1. |
| VEX.NDS.128.F2.0F.WIG 5F /r<br>VMAXSD xmm1, xmm2,<br>xmm3/m64              | RVM        | V/V                          | AVX                      | Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2. |
| EVEX.NDS.LIG.F2.0F.W1 5F /r<br>VMAXSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{sae} | T1S        | V/V                          | AVX512F                  | Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2. |

### 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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

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

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

**Operation**

```

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

```

**VMAXSD (EVEX encoded version)**

```

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

```

**VMAXSD (VEX.128 encoded version)**

```

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

```

**MAXSD (128-bit Legacy SSE version)**

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMAXSD __m128d __mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d __mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d __mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d __mm_max_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E3.

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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| F3 0F 5F /r<br>MAXSS xmm1, xmm2/m32  | RM        | V/V                          | SSE                      | Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1. |
| VEX.NDS.128.F3.0F.WIG 5F /r<br>VMAXSS xmm1, xmm2,<br>xmm3/m32              | RVM       | V/V                          | AVX                      | Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2. |
| EVEX.NDS.LIG.F3.0F.W0 5F /r<br>VMAXSS xmm1 {k1}{z},<br>xmm2, xmm3/m32{sae} | T1S       | V/V                          | AVX512F                  | Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2. |

**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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

**Description**

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

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

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

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

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

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

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

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



**Operation**

```

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

```

**VMAXSS (EVEX encoded version)**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMAXSS (VEX.128 encoded version)**

```

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

```

**MAXSS (128-bit Legacy SSE version)**

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMAXSS __m128_mm_max_round_ss(__m128 a, __m128 b, int);
VMAXSS __m128_mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128_mm_maskz_max_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128_mm_max_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

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

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

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

### Operation

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

### VMINPD (EVEX encoded version)

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

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

        THEN

          DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[63:0])

        ELSE

          DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[i+63:i])

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE DEST[i+63:i] ← 0 ; zeroing-masking

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VMINPD (VEX.256 encoded version)

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])

DEST[191:128] ← MIN(SRC1[191:128], SRC2[191:128])

DEST[255:192] ← MIN(SRC1[255:192], SRC2[255:192])

### VMINPD (VEX.128 encoded version)

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])

DEST[MAX\_VL-1:128] ← 0

**MINPD (128-bit Legacy SSE version)**

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMINPD \_\_m512d \_\_mm512\_min\_pd(\_\_m512d a, \_\_m512d b);

VMINPD \_\_m512d \_\_mm512\_mask\_min\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VMINPD \_\_m512d \_\_mm512\_maskz\_min\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VMINPD \_\_m512d \_\_mm512\_min\_round\_pd(\_\_m512d a, \_\_m512d b, int);

VMINPD \_\_m512d \_\_mm512\_mask\_min\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMINPD \_\_m512d \_\_mm512\_maskz\_min\_round\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMINPD \_\_m256d \_\_mm256\_mask\_min\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VMINPD \_\_m256d \_\_mm256\_maskz\_min\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VMINPD \_\_m128d \_\_mm\_mask\_min\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VMINPD \_\_m128d \_\_mm\_maskz\_min\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VMINPD \_\_m256d \_\_mm256\_min\_pd(\_\_m256d a, \_\_m256d b);

MINPD \_\_m128d \_\_mm\_min\_pd(\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E2.

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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| OF 5D /r<br>MINPS xmm1, xmm2/m128  | RM        | V/V                          | SSE                      | Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.  |
| VEX.NDS.128.OF.WIG 5D /r<br>VMINPS xmm1, xmm2,<br>xmm3/m128                      | RVM       | V/V                          | AVX                      | Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.  |
| VEX.NDS.256.OF.WIG 5D /r<br>VMINPS ymm1, ymm2,<br>ymm3/m256                      | RVM       | V/V                          | AVX                      | Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.   |
| EVEX.NDS.128.OF.W0 5D /r<br>VMINPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1. |
| EVEX.NDS.256.OF.W0 5D /r<br>VMINPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | FV        | V/V                          | AVX512VL<br>AVX512F      | Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1. |
| EVEX.NDS.512.OF.W0 5D /r<br>VMINPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{sae} | FV        | V/V                          | AVX512F                  | Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

### Operation

MIN(SRC1, SRC2)

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

### VMINPS (EVEX encoded version)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

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

        THEN

          DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[31:0])

        ELSE

          DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[i+31:i])

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VMINPS (VEX.256 encoded version)

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])

DEST[159:128] ← MIN(SRC1[159:128], SRC2[159:128])

DEST[191:160] ← MIN(SRC1[191:160], SRC2[191:160])

DEST[223:192] ← MIN(SRC1[223:192], SRC2[223:192])

DEST[255:224] ← MIN(SRC1[255:224], SRC2[255:224])

**VMINPS (VEX.128 encoded version)**

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

**MINPS (128-bit Legacy SSE version)**

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

**Intel C/C++ Compiler Intrinsic Equivalent**

VMINPS \_\_m512 \_\_mm512\_min\_ps( \_\_m512 a, \_\_m512 b);  
 VMINPS \_\_m512 \_\_mm512\_mask\_min\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMINPS \_\_m512 \_\_mm512\_maskz\_min\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMINPS \_\_m512 \_\_mm512\_min\_round\_ps( \_\_m512 a, \_\_m512 b, int);  
 VMINPS \_\_m512 \_\_mm512\_mask\_min\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMINPS \_\_m512 \_\_mm512\_maskz\_min\_round\_ps( \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMINPS \_\_m256 \_\_mm256\_mask\_min\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMINPS \_\_m256 \_\_mm256\_maskz\_min\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMINPS \_\_m128 \_\_mm\_mask\_min\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMINPS \_\_m128 \_\_mm\_maskz\_min\_ps( \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMINPS \_\_m256 \_\_mm256\_min\_ps( \_\_m256 a, \_\_m256 b);  
 MINPS \_\_m128 \_\_mm\_min\_ps( \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E2.

## MINSND—Return Minimum Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| F2 0F 5D /r<br>MINSND xmm1, xmm2/m64  | RM         | V/V                          | SSE2                     | Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1. |
| VEX.NDS.128.F2.0F.WIG 5D /r<br>VMINSND xmm1, xmm2, xmm3/m64                 | RVM        | V/V                          | AVX                      | Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2. |
| EVEX.NDS.LIG.F2.0F.W1 5D /r<br>VMINSND xmm1 {k1}{z}, xmm2,<br>xmm3/m64{sae} | T1S        | V/V                          | AVX512F                  | Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2. |

### 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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

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

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

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



**Operation**

```

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

```

**MINSD (EVEX encoded version)**

```

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

```

**MINSD (VEX.128 encoded version)**

```

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

```

**MINSD (128-bit Legacy SSE version)**

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINSD __m128d __mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d __mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d __mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d __mm_min_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

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

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| F3 0F 5D /r<br>MINSS xmm1,xmm2/m32   | RM         | V/V                          | SSE                      | Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1. |
| VEX.NDS.128.F3.0F.WIG 5D /r<br>VMINSS xmm1,xmm2, xmm3/m32                  | RVM        | V/V                          | AVX                      | Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2. |
| EVEX.NDS.LIG.F3.0F.W0 5D /r<br>VMINSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae} | T1S        | V/V                          | AVX512F                  | Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2. |

### 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      | ModRM:r/m (r) | NA        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

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

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

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

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

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

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

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

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

**Operation**

```

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

```

**MINSS (EVEX encoded version)**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMINSS (VEX.128 encoded version)**

```

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

```

**MINSS (128-bit Legacy SSE version)**

```

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

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

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

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

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

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

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

**Instruction Operand Encoding**

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

## Description

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

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

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

EVEX.512 encoded version:

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

VEX.256 and EVEX.256 encoded versions:

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

128-bit versions:

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

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

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

## Operation

### VMOVAPD (EVEX encoded versions, register-copy form)

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ← SRC[i+63:i]

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE DEST[i+63:i] ← 0 ; zeroing-masking

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVAPD (EVEX encoded versions, store-form)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVAPD (EVEX encoded versions, load-form)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

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

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

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

DEST[255:0] ← SRC[255:0]

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

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

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

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVAPD \_\_m512d \_\_mm512\_load\_pd( void \* m);

VMOVAPD \_\_m512d \_\_mm512\_mask\_load\_pd(\_\_m512d s, \_\_mmask8 k, void \* m);

VMOVAPD \_\_m512d \_\_mm512\_maskz\_load\_pd( \_\_mmask8 k, void \* m);

VMOVAPD void \_\_mm512\_store\_pd( void \* d, \_\_m512d a);

VMOVAPD void \_\_mm512\_mask\_store\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VMOVAPD \_\_m256d \_\_mm256\_mask\_load\_pd(\_\_m256d s, \_\_mmask8 k, void \* m);

VMOVAPD \_\_m256d \_\_mm256\_maskz\_load\_pd( \_\_mmask8 k, void \* m);

VMOVAPD void \_\_mm256\_mask\_store\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

```
VMOVAPD __m128d _mm_mask_load_pd(__m128d s, __mmask8 k, void * m);  
VMOVAPD __m128d _mm_maskz_load_pd( __mmask8 k, void * m);  
VMOVAPD void _mm_mask_store_pd( void * d, __mmask8 k, __m128d a);  
MOVAPD __m256d _mm256_load_pd (double * p);  
MOVAPD void _mm256_store_pd(double * p, __m256d a);  
MOVAPD __m128d _mm_load_pd (double * p);  
MOVAPD void _mm_store_pd(double * p, __m128d a);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

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

EVEX-encoded instruction, see Exceptions Type E1.

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

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

| Opcode/<br>Instruction                                  | Op/En  | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|--------|------------------------------|--------------------------|---|
| OF 28 /r<br>MOVAPS xmm1, xmm2/m128                      | RM     | V/V                          | SSE                      | Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.                     |
| OF 29 /r<br>MOVAPS xmm2/m128, xmm1                      | MR     | V/V                          | SSE                      | Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.                     |
| VEX.128.OF.WIG 28 /r<br>VMOVAPS xmm1, xmm2/m128         | RM     | V/V                          | AVX                      | Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.                     |
| VEX.128.OF.WIG 29 /r<br>VMOVAPS xmm2/m128, xmm1         | MR     | V/V                          | AVX                      | Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.                     |
| VEX.256.OF.WIG 28 /r<br>VMOVAPS ymm1, ymm2/m256         | RM     | V/V                          | AVX                      | Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.                     |
| VEX.256.OF.WIG 29 /r<br>VMOVAPS ymm2/m256, ymm1         | MR     | V/V                          | AVX                      | Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.                     |
| EVEX.128.OF.W0 28 /r<br>VMOVAPS xmm1 {k1}{z}, xmm2/m128 | FVM-RM | V/V                          | AVX512VL<br>AVX512F      | Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.OF.W0 28 /r<br>VMOVAPS ymm1 {k1}{z}, ymm2/m256 | FVM-RM | V/V                          | AVX512VL<br>AVX512F      | Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.OF.W0 28 /r<br>VMOVAPS zmm1 {k1}{z}, zmm2/m512 | FVM-RM | V/V                          | AVX512F                  | Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |
| EVEX.128.OF.W0 29 /r<br>VMOVAPS xmm2/m128 {k1}{z}, xmm1 | FVM-MR | V/V                          | AVX512VL<br>AVX512F      | Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1. |
| EVEX.256.OF.W0 29 /r<br>VMOVAPS ymm2/m256 {k1}{z}, ymm1 | FVM-MR | V/V                          | AVX512VL<br>AVX512F      | Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1. |
| EVEX.512.OF.W0 29 /r<br>VMOVAPS zmm2/m512 {k1}{z}, zmm1 | FVM-MR | V/V                          | AVX512F                  | Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1. |

### Instruction Operand Encoding

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

### Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit,



256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

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

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

EVEX.512 encoded version:

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

VEX.256 and EVEX.256 encoded version:

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

128-bit versions:

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

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

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

## Operation

### VMOVAPS (EVEX encoded versions, register-copy form)

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

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

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

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVAPS (EVEX encoded versions, store-form)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

SRC[i+31:i]

ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVAPS (EVEX encoded versions, load-form)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

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

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

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

DEST[255:0] ← SRC[255:0]

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

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

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

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVAPS \_\_m512 \_\_mm512\_load\_ps( void \* m);

VMOVAPS \_\_m512 \_\_mm512\_mask\_load\_ps(\_\_m512 s, \_\_mmask16 k, void \* m);

VMOVAPS \_\_m512 \_\_mm512\_maskz\_load\_ps( \_\_mmask16 k, void \* m);

VMOVAPS void \_\_mm512\_store\_ps( void \* d, \_\_m512 a);

VMOVAPS void \_\_mm512\_mask\_store\_ps( void \* d, \_\_mmask16 k, \_\_m512 a);

VMOVAPS \_\_m256 \_\_mm256\_mask\_load\_ps(\_\_m256 a, \_\_mmask8 k, void \* s);

VMOVAPS \_\_m256 \_\_mm256\_maskz\_load\_ps( \_\_mmask8 k, void \* s);

VMOVAPS void \_\_mm256\_mask\_store\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VMOVAPS \_\_m128 \_\_mm\_mask\_load\_ps(\_\_m128 a, \_\_mmask8 k, void \* s);

```
VMOVAPS __m128 _mm_maskz_load_ps( __mmask8 k, void * s);  
VMOVAPS void _mm_mask_store_ps( void * d, __mmask8 k, __m128 a);  
MOVAPS __m256 _mm256_load_ps( float * p);  
MOVAPS void _mm256_store_ps( float * p, __m256 a);  
MOVAPS __m128 _mm_load_ps( float * p);  
MOVAPS void _mm_store_ps( float * p, __m128 a);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 1.SSE; additionally

#UD                    If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

**MOVD/MOVQ—Move Doubleword and Quadword**

| Opcode/<br>Instruction                         | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description                                  |
|--|---------|------------------------------|--------------------------|--|
| 66 0F 6E /r<br>MOVD xmm1, r32/m32              | MR      | V/V                          | SSE2                     | Move doubleword from r/m32 to xmm1.          |
| 66 REX.W 0F 6E /r<br>MOVQ xmm1, r64/m64        | MR      | V/N.E.                       | SSE2                     | Move quadword from r/m64 to xmm1.            |
| VEX.128.66.0F.W0 6E /r<br>VMOVD xmm1, r32/m32  | MR      | V/V                          | AVX                      | Move doubleword from r/m32 to xmm1.          |
| VEX.128.66.0F.W1 6E /r<br>VMOVQ xmm1, r64/m64  | MR      | V/N.E. <sup>1</sup>          | AVX                      | Move quadword from r/m64 to xmm1.            |
| EVEX.128.66.0F.W0 6E /r<br>VMOVD xmm1, r32/m32 | T1S-RM  | V/V                          | AVX512F                  | Move doubleword from r/m32 to xmm1.          |
| EVEX.128.66.0F.W1 6E /r<br>VMOVQ xmm1, r64/m64 | T1S-RM  | V/N.E. <sup>1</sup>          | AVX512F                  | Move quadword from r/m64 to xmm1.            |
| 66 0F 7E /r<br>MOVD r32/m32, xmm1              | MR      | V/V                          | SSE2                     | Move doubleword from xmm1 register to r/m32. |
| 66 REX.W 0F 7E /r<br>MOVQ r64/m64, xmm1        | MR      | V/N.E.                       | SSE2                     | Move quadword from xmm1 register to r/m64.   |
| VEX.128.66.0F.W0 7E /r<br>VMOVD r32/m32, xmm1  | MR      | V/V                          | AVX                      | Move doubleword from xmm1 register to r/m32. |
| VEX.128.66.0F.W1 7E /r<br>VMOVQ r64/m64, xmm1  | MR      | V/N.E. <sup>1</sup>          | AVX                      | Move quadword from xmm1 register to r/m64.   |
| EVEX.128.66.0F.W0 7E /r<br>VMOVD r32/m32, xmm1 | T1S-MR  | V/V                          | AVX512F                  | Move doubleword from xmm1 register to r/m32. |
| EVEX.128.66.0F.W1 7E /r<br>VMOVQ r64/m64, xmm1 | T1S-MR  | V/N.E. <sup>1</sup>          | AVX512F                  | Move quadword from xmm1 register to r/m64.   |

**NOTES:**

- For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Instruction Operand Encoding**

| Op/En  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|--------|---------------|---------------|-----------|-----------|
| RM     | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| MR     | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| T1S-RM | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1S-MR | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

## Description

### MOVD/Q with XMM destination:

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

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

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

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

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

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

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

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

## Operation

### **MOVD (Legacy SSE version when destination is an XMM register)**

DEST[31:0] ← SRC[31:0]  
 DEST[127:32] ← 0H  
 DEST[MAX\_VL-1:128] (Unmodified)

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

DEST[31:0] ← SRC[31:0]  
 DEST[MAX\_VL-1:32] ← 0H

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

DEST[31:0] ← SRC[31:0]  
 DEST[511:32] ← 0H

### **MOVQ (Legacy SSE version when destination is an XMM register)**

DEST[63:0] ← SRC[63:0]  
 DEST[127:64] ← 0H  
 DEST[MAX\_VL-1:128] (Unmodified)

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

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0H

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

DEST[63:0] ← SRC[63:0]  
 DEST[511:64] ← 0H

### **MOVD / VMOVD (when destination is not an XMM register)**

DEST[31:0] ← SRC[31:0]

### **MOVQ / VMOVQ (when destination is not an XMM register)**

DEST[63:0] ← SRC[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVD __m128i _mm_cvtsi32_si128( int);
VMOVD int _mm_cvtsi128_si32( __m128i );
VMOVQ __m128i _mm_cvtsi64_si128 ( __int64);
VMOVQ __int64 _mm_cvtsi128_si64(__m128i );
VMOVQ __m128i _mm_loadl_epi64( __m128i * s);
VMOVQ void _mm_storel_epi64( __m128i * d, __m128i s);
MOVD __m128i _mm_cvtsi32_si128(int a)
MOVD int _mm_cvtsi128_si32(__m128i a)
MOVQ __m128i _mm_cvtsi64_si128(__int64 a)
MOVQ __int64 _mm_cvtsi128_si64(__m128i a)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E9NF.

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

## MOVQ—Move Quadword

| Opcode<br>Instruction                           | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description                                   |
|---|---------|------------------------------|--------------------------|---|
| F3 0F 7E /r<br>MOVQ xmm1, xmm2/m64              | RM      | V/V                          | SSE2                     | Move quadword from xmm2/m64 to xmm1.          |
| VEX.128.F3.0F.WIG 7E /r<br>VMOVQ xmm1, xmm2/m64 | RM      | V/V                          | AVX                      | Move quadword from xmm2/m64 to xmm1.          |
| EVEX.128.F3.0F.W1 7E /r<br>VMOVQ xmm1, xmm2/m64 | T1S-RM  | V/V                          | AVX512F                  | Move quadword from xmm2/m64 to xmm1.          |
| 66 0F D6 /r<br>MOVQ xmm1/m64, xmm2              | MR      | V/V                          | SSE2                     | Move quadword from xmm2 register to xmm1/m64. |
| VEX.128.66.0F.WIG D6 /r<br>VMOVQ xmm1/m64, xmm2 | MR      | V/V                          | AVX                      | Move quadword from xmm2 register to xmm1/m64. |
| EVEX.128.66.0F.W1 D6 /r<br>VMOVQ xmm1/m64, xmm2 | T1S-MR  | V/V                          | AVX512F                  | Move quadword from xmm2 register to xmm1/m64. |

### Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|--------|---------------|---------------|-----------|-----------|
| RM     | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| MR     | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| T1S-RM | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| T1S-MR | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

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

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

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

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

**Operation****MOVQ (F3 0F 7E and 66 0F D6) with XMM register source and destination:**

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

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

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

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

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

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

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

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

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

**MOVQ (7E) with memory source:**

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

**VMOVQ (7E - VEX.128 encoded version) with memory source:**

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

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

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

**MOVQ (D6) with memory dest:**

DEST[63:0] ← SRC[63:0]

**VMOVQ (D6) with memory dest:**

DEST[63:0] ← SRC2[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVQ \_\_m128i \_mm\_loadu\_si64( void \* s);  
 VMOVQ void \_mm\_storeu\_si64( void \* d, \_\_m128i s);  
 MOVQ \_\_m128i \_mm\_move\_epi64(\_\_m128i a);

**SIMD Floating-Point Exceptions**

None



**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E9NF.

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

**MOVDDUP—Replicate Double FP Values**

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

**Instruction Operand Encoding**

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

**Description**

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

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

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

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

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

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

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

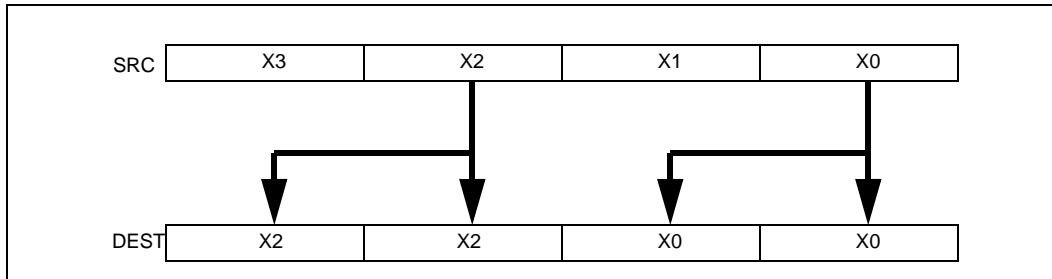


Figure 5-26. VMOVDDUP Operation

**Operation****VMOVDDUP (EVEX encoded versions)**

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

TMP\_SRC[63:0] ← SRC[63:0]

TMP\_SRC[127:64] ← SRC[63:0]

IF VL ≥ 256

    TMP\_SRC[191:128] ← SRC[191:128]

    TMP\_SRC[255:192] ← SRC[191:128]

FI;

IF VL ≥ 512

    TMP\_SRC[319:256] ← SRC[319:256]

    TMP\_SRC[383:320] ← SRC[319:256]

    TMP\_SRC[477:384] ← SRC[477:384]

    TMP\_SRC[511:484] ← SRC[477:384]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_SRC[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

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

        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVDDUP (VEX.256 encoded version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[191:128] ← SRC[191:128]

DEST[255:192] ← SRC[191:128]

DEST[MAX\_VL-1:256] ← 0

**VMOVDDUP (VEX.128 encoded version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[MAX\_VL-1:128] ← 0

**MOVDDUP (128-bit Legacy SSE version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVDDUP \_\_m512d \_\_mm512\_movedup\_pd(\_\_m512d a);

VMOVDDUP \_\_m512d \_\_mm512\_mask\_movedup\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VMOVDDUP \_\_m512d \_\_mm512\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m512d a);

VMOVDDUP \_\_m256d \_\_mm256\_mask\_movedup\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VMOVDDUP \_\_m256d \_\_mm256\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m256d a);

VMOVDDUP \_\_m128d \_\_mm\_mask\_movedup\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VMOVDDUP \_\_m128d \_\_mm\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m128d a);

MOVDDUP \_\_m256d \_\_mm256\_movedup\_pd(\_\_m256d a);

MOVDDUP \_\_m128d \_\_mm\_movedup\_pd(\_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E5NF.

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

**MOVDQA, VMOVDQA32/64—Move Aligned Packed Integer Values**

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

| Opcode/<br>Instruction  | Op/En  | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|--------|------------------------------|--------------------------|--|
| EVEX.512.66.0F.W1 6F /r<br>VMOVDQA64 zmm1 {k1}{z},<br>zmm2/m512 | FVM-RM | V/V                          | AVX512F                  | Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1. |
| EVEX.128.66.0F.W1 7F /r<br>VMOVDQA64 xmm2/m128 {k1}{z},<br>xmm1 | FVM-MR | V/V                          | AVX512VL<br>AVX512F      | Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1. |
| EVEX.256.66.0F.W1 7F /r<br>VMOVDQA64 ymm2/m256 {k1}{z},<br>ymm1 | FVM-MR | V/V                          | AVX512VL<br>AVX512F      | Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1. |
| EVEX.512.66.0F.W1 7F /r<br>VMOVDQA64 zmm2/m512 {k1}{z},<br>zmm1 | FVM-MR | V/V                          | AVX512F                  | Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1. |

### Instruction Operand Encoding

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

### Description

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

EVEX encoded versions:

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

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

VEX.256 encoded version:

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

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

128-bit versions:

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

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

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

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

### Operation

#### VMOVDQA32 (EVEX encoded versions, register-copy form)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VMOVDQA32 (EVEX encoded versions, store-form)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

#### VMOVDQA32 (EVEX encoded versions, load-form)

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVDQA64 (EVEX encoded versions, register-copy form)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVDQA64 (EVEX encoded versions, store-form)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQA64 (EVEX encoded versions, load-form)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

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

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

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

DEST[255:0] ← SRC[255:0]

**VMOVDQA (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)



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

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVDQA32 __m512i __mm512_load_epi32( void * sa);
VMOVDQA32 __m512i __mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i __mm512_maskz_load_epi32( __mmask16 k, void * sa);
VMOVDQA32 void __mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void __mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i __mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i __mm256_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void __mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void __mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i __mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i __mm_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void __mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void __mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i __mm512_load_epi64( void * sa);
VMOVDQA64 __m512i __mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i __mm512_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void __mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i __mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i __mm256_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void __mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i __mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i __mm_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void __mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i __mm256_load_si256 (__m256i * p);
MOVDQA __m256i __mm256_store_si256(__m256i *p, __m256i a);
MOVDQA __m128i __mm_load_si128 (__m128i * p);
MOVDQA void __mm_store_si128(__m128i *p, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

EVEX-encoded instruction, see Exceptions Type E1.

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

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

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

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

### Instruction Operand Encoding

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

#### Description

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

#### EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector

register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

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

**VEX.256 encoded version:**

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

Bits (MAX\_VL-1:256) of the destination register are zeroed.

**128-bit versions:**

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

**128-bit Legacy SSE version:** Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

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

**VEX.128 encoded version:** Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation**

**VMOVDQU8 (EVEX encoded versions, register-copy form)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SRC[i+7:i]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE DEST[i+7:i] ← 0 ; zeroing-masking

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVDQU8 (EVEX encoded versions, store-form)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ←

      SRC[i+7:i]

  ELSE \*DEST[i+7:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

**VMOVDQU8 (EVEX encoded versions, load-form)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SRC[i+7:i]

```

ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
    ELSE DEST[i+7:i] ← 0                ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU16 (EVEX encoded versions, register-copy form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE DEST[i+15:i] ← 0                ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU16 (EVEX encoded versions, store-form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ←
            SRC[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR;

```

**VMOVDQU16 (EVEX encoded versions, load-form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE DEST[i+15:i] ← 0                ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU32 (EVEX encoded versions, register-copy form)**

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

```

FOR j ← 0 TO KL-1
    i ← j * 32

```

```

IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] ← SRC[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE DEST[i+31:i] ← 0 ; zeroing-masking
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU32 (EVEX encoded versions, store-form)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      SRC[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;

```

**VMOVDQU32 (EVEX encoded versions, load-form)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] ← 0 ; zeroing-masking
      FI
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU64 (EVEX encoded versions, register-copy form)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] ← 0 ; zeroing-masking
      FI
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU64 (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking

  FI;
ENDFOR;

```

**VMOVDQU64 (EVEX encoded versions, load-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0 ; zeroing-masking
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMOVDQU (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVDQU (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

VMOVDQU (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**VMOVDQU (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVDQU (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVDQU16 \_\_m512i \_\_mm512\_mask\_loadu\_epi16(\_\_m512i s, \_\_mmask32 k, void \* sa);

VMOVDQU16 \_\_m512i \_\_mm512\_maskz\_loadu\_epi16(\_\_mmask32 k, void \* sa);

VMOVDQU16 void \_\_mm512\_mask\_storeu\_epi16(void \* d, \_\_mmask32 k, \_\_m512i a);

VMOVDQU16 \_\_m256i \_\_mm256\_mask\_loadu\_epi16(\_\_m256i s, \_\_mmask16 k, void \* sa);

VMOVDQU16 \_\_m256i \_\_mm256\_maskz\_loadu\_epi16(\_\_mmask16 k, void \* sa);

VMOVDQU16 void \_\_mm256\_mask\_storeu\_epi16(void \* d, \_\_mmask16 k, \_\_m256i a);

VMOVDQU16 void \_\_mm256\_maskz\_storeu\_epi16(void \* d, \_\_mmask16 k);

VMOVDQU16 \_\_m128i \_\_mm\_mask\_loadu\_epi16(\_\_m128i s, \_\_mmask8 k, void \* sa);

VMOVDQU16 \_\_m128i \_\_mm\_maskz\_loadu\_epi16(\_\_mmask8 k, void \* sa);

VMOVDQU16 void \_\_mm\_mask\_storeu\_epi16(void \* d, \_\_mmask8 k, \_\_m128i a);

VMOVDQU32 \_\_m512i \_\_mm512\_loadu\_epi32( void \* sa);

VMOVDQU32 \_\_m512i \_mm512\_mask\_loadu\_epi32(\_\_m512i s, \_\_mmask16 k, void \* sa);  
 VMOVDQU32 \_\_m512i \_mm512\_maskz\_loadu\_epi32(\_\_mmask16 k, void \* sa);  
 VMOVDQU32 void \_mm512\_storeu\_epi32(void \* d, \_\_m512i a);  
 VMOVDQU32 void \_mm512\_mask\_storeu\_epi32(void \* d, \_\_mmask16 k, \_\_m512i a);  
 VMOVDQU32 \_\_m256i \_mm256\_mask\_loadu\_epi32(\_\_m256i s, \_\_mmask8 k, void \* sa);  
 VMOVDQU32 \_\_m256i \_mm256\_maskz\_loadu\_epi32(\_\_mmask8 k, void \* sa);  
 VMOVDQU32 void \_mm256\_storeu\_epi32(void \* d, \_\_m256i a);  
 VMOVDQU32 void \_mm256\_mask\_storeu\_epi32(void \* d, \_\_mmask8 k, \_\_m256i a);  
 VMOVDQU32 \_\_m128i \_mm\_mask\_loadu\_epi32(\_\_m128i s, \_\_mmask8 k, void \* sa);  
 VMOVDQU32 \_\_m128i \_mm\_maskz\_loadu\_epi32(\_\_mmask8 k, void \* sa);  
 VMOVDQU32 void \_mm\_storeu\_epi32(void \* d, \_\_m128i a);  
 VMOVDQU32 void \_mm\_mask\_storeu\_epi32(void \* d, \_\_mmask8 k, \_\_m128i a);  
 VMOVDQU64 \_\_m512i \_mm512\_loadu\_epi64(void \* sa);  
 VMOVDQU64 \_\_m512i \_mm512\_mask\_loadu\_epi64(\_\_m512i s, \_\_mmask8 k, void \* sa);  
 VMOVDQU64 \_\_m512i \_mm512\_maskz\_loadu\_epi64(\_\_mmask8 k, void \* sa);  
 VMOVDQU64 void \_mm512\_storeu\_epi64(void \* d, \_\_m512i a);  
 VMOVDQU64 void \_mm512\_mask\_storeu\_epi64(void \* d, \_\_mmask8 k, \_\_m512i a);  
 VMOVDQU64 \_\_m256i \_mm256\_mask\_loadu\_epi64(\_\_m256i s, \_\_mmask8 k, void \* sa);  
 VMOVDQU64 \_\_m256i \_mm256\_maskz\_loadu\_epi64(\_\_mmask8 k, void \* sa);  
 VMOVDQU64 void \_mm256\_storeu\_epi64(void \* d, \_\_m256i a);  
 VMOVDQU64 void \_mm256\_mask\_storeu\_epi64(void \* d, \_\_mmask8 k, \_\_m256i a);  
 VMOVDQU64 \_\_m128i \_mm\_mask\_loadu\_epi64(\_\_m128i s, \_\_mmask8 k, void \* sa);  
 VMOVDQU64 \_\_m128i \_mm\_maskz\_loadu\_epi64(\_\_mmask8 k, void \* sa);  
 VMOVDQU64 void \_mm\_storeu\_epi64(void \* d, \_\_m128i a);  
 VMOVDQU64 void \_mm\_mask\_storeu\_epi64(void \* d, \_\_mmask8 k, \_\_m128i a);  
 VMOVDQU8 \_\_m512i \_mm512\_mask\_loadu\_epi8(\_\_m512i s, \_\_mmask64 k, void \* sa);  
 VMOVDQU8 \_\_m512i \_mm512\_maskz\_loadu\_epi8(\_\_mmask64 k, void \* sa);  
 VMOVDQU8 void \_mm512\_mask\_storeu\_epi8(void \* d, \_\_mmask64 k, \_\_m512i a);  
 VMOVDQU8 \_\_m256i \_mm256\_mask\_loadu\_epi8(\_\_m256i s, \_\_mmask32 k, void \* sa);  
 VMOVDQU8 \_\_m256i \_mm256\_maskz\_loadu\_epi8(\_\_mmask32 k, void \* sa);  
 VMOVDQU8 void \_mm256\_mask\_storeu\_epi8(void \* d, \_\_mmask32 k, \_\_m256i a);  
 VMOVDQU8 void \_mm256\_maskz\_storeu\_epi8(void \* d, \_\_mmask32 k);  
 VMOVDQU8 \_\_m128i \_mm\_mask\_loadu\_epi8(\_\_m128i s, \_\_mmask16 k, void \* sa);  
 VMOVDQU8 \_\_m128i \_mm\_maskz\_loadu\_epi8(\_\_mmask16 k, void \* sa);  
 VMOVDQU8 void \_mm\_mask\_storeu\_epi8(void \* d, \_\_mmask16 k, \_\_m128i a);  
 MOVDDQU \_\_m256i \_mm256\_loadu\_si256(\_\_m256i \* p);  
 MOVDDQU \_\_m256i \_mm256\_storeu\_si256(\_\_m256i \* p, \_\_m256i a);  
 MOVDDQU \_\_m128i \_mm\_loadu\_si128(\_\_m128i \* p);  
 MOVDDQU \_\_m128i \_mm\_storeu\_si128(\_\_m128i \* p, \_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## MOVHPLS—Move Packed Single-Precision Floating-Point Values High to Low

| Opcode/<br>Instruction                                | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| OF 12 /r<br>MOVHPLS xmm1, xmm2                        | RM         | V/V                          | SSE                      | Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.   |
| VEX.NDS.128.OF.WIG 12 /r<br>VMOVHPLS xmm1, xmm2, xmm3 | RVM        | V/V                          | AVX                      | Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2. |
| EVEX.NDS.128.OF.W0 12 /r<br>VMOVHPLS xmm1, xmm2, xmm3 | RVM        | V/V                          | AVX512F                  | Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2. |

### Instruction Operand Encoding<sup>1</sup>

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w) | vvvv (r)      | ModRM:r/m (r) | NA        |

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

#### 128-bit and EVEX three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

If VMOVHPLS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### MOVHPLS (128-bit two-argument form)

DEST[63:0] ← SRC[127:64]  
DEST[MAX\_VL-1:64] (Unmodified)

#### VMOVHPLS (128-bit three-argument form - VEX & EVEX)

DEST[63:0] ← SRC2[127:64]  
DEST[127:64] ← SRC1[127:64]  
DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

MOVHPLS \_\_m128 \_\_mm\_movehl\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally  
#UD                    If VEX.L = 1.  
EVEX-encoded instruction, see Exceptions Type E7NM.128.

## MOVHPD—Move High Packed Double-Precision Floating-Point Value

| Opcode/<br>Instruction                                 | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|---------|------------------------------|--------------------------|--|
| 66 0F 16 /r<br>MOVHPD xmm1, m64                        | RM      | V/V                          | SSE2                     | Move double-precision floating-point value from m64 to high quadword of xmm1.      |
| VEX.NDS.128.66.0F.WIG 16 /r<br>VMOVHPD xmm2, xmm1, m64 | RVM     | V/V                          | AVX                      | Merge double-precision floating-point value from m64 and the low quadword of xmm1. |
| EVEX.NDS.128.66.0F.W1 16 /r<br>VMOVHPD xmm2, xmm1, m64 | T1S     | V/V                          | AVX512F                  | Merge double-precision floating-point value from m64 and the low quadword of xmm1. |
| 66 0F 17 /r<br>MOVHPD m64, xmm1                        | MR      | V/V                          | SSE2                     | Move double-precision floating-point value from high quadword of xmm1 to m64.      |
| VEX.128.66.0F.WIG 17 /r<br>VMOVHPD m64, xmm1           | MR      | V/V                          | AVX                      | Move double-precision floating-point value from high quadword of xmm1 to m64.      |
| EVEX.128.66.0F.W1 17 /r<br>VMOVHPD m64, xmm1           | T1S-MR  | V/V                          | AVX512F                  | Move double-precision floating-point value from high quadword of xmm1 to m64.      |

### 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      | ModRM:r/m (r) | NA        |
| MR     | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |
| T1S    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |
| T1S-MR | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### **MOVHPD (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)  
DEST[127:64] ← SRC[63:0]  
DEST[MAX\_VL-1:128] (Unmodified)

#### **VMOVHPD (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC1[63:0]  
DEST[127:64] ← SRC2[63:0]  
DEST[MAX\_VL-1:128] ← 0

#### **VMOVHPD (store)**

DEST[63:0] ← SRC[127:64]

### Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD \_\_m128d \_mm\_loadh\_pd ( \_\_m128d a, double \*p)  
MOVHPD void \_mm\_storeh\_pd (double \*p, \_\_m128d a)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD                    If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVHPS—Move High Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction                              | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|---------|------------------------------|--------------------------|--|
| 0F 16 /r<br>MOVHPS xmm1, m64                        | RM      | V/V                          | SSE                      | Move two packed single-precision floating-point values from m64 to high quadword of xmm1.      |
| VEX.NDS.128.0F.WIG 16 /r<br>VMOVHPS xmm2, xmm1, m64 | RVM     | V/V                          | AVX                      | Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1. |
| EVEX.NDS.128.0F.W0 16 /r<br>VMOVHPS xmm2, xmm1, m64 | T2      | V/V                          | AVX512F                  | Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1. |
| 0F 17 /r<br>MOVHPS m64, xmm1                        | MR      | V/V                          | SSE                      | Move two packed single-precision floating-point values from high quadword of xmm1 to m64.      |
| VEX.128.0F.WIG 17 /r<br>VMOVHPS m64, xmm1           | MR      | V/V                          | AVX                      | Move two packed single-precision floating-point values from high quadword of xmm1 to m64.      |
| EVEX.128.0F.W0 17 /r<br>VMOVHPS m64, xmm1           | T2-MR   | V/V                          | AVX512F                  | Move two packed single-precision floating-point values from high quadword of xmm1 to m64.      |

### 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      | ModRM:r/m (r) | NA        |
| MR    | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |
| T2    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |
| T2-MR | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### **MOVHPS (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)  
DEST[127:64] ← SRC[63:0]  
DEST[MAX\_VL-1:128] (Unmodified)

#### **VMOVHPS (VEX.128 and EVEX encoded load)**

DEST[63:0] ← SRC1[63:0]  
DEST[127:64] ← SRC2[63:0]  
DEST[MAX\_VL-1:128] ← 0

#### **VMOVHPS (store)**

DEST[63:0] ← SRC[127:64]

### Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS \_\_m128 \_mm\_loadh\_pi (\_\_m128 a, \_\_m64 \*p)  
MOVHPS void \_mm\_storeh\_pi (\_\_m64 \*p, \_\_m128 a)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD                      If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

| Opcode/<br>Instruction                                | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| OF 16 /r<br>MOVLHPS xmm1, xmm2                        | RM        | V/V                          | SSE                      | Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.  |
| VEX.NDS.128.OF.WIG 16 /r<br>VMOVLHPS xmm1, xmm2, xmm3 | RVM       | V/V                          | AVX                      | Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2. |
| EVEX.NDS.128.OF.W0 16 /r<br>VMOVLHPS xmm1, xmm2, xmm3 | RVM       | V/V                          | AVX512F                  | Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2. |

### Instruction Operand Encoding<sup>1</sup>

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w) | vvvv (r)      | ModRM:r/m (r) | NA        |

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAX\_VL-1:128) of the corresponding destination register are unmodified.

#### 128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L'L = 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L = 1 will cause an #UD exception.

### Operation

#### MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)  
 DEST[127:64] ← SRC[63:0]  
 DEST[MAX\_VL-1:128] (Unmodified)

#### VMOVLHPS (128-bit three-argument form - VEX & EVEX)

DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS \_\_m128 \_\_mm\_movelh\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally  
#UD                    If VEX.L = 1.  
EVEX-encoded instruction, see Exceptions Type E7NM.128.



## MOVLPD—Move Low Packed Double-Precision Floating-Point Value

| Opcode/<br>Instruction                                 | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|------------------------------|--------------------------|---|
| 66 0F 12 /r<br>MOVLPD xmm1, m64                        | RM      | V/V                          | SSE2                     | Move double-precision floating-point value from m64 to low quadword of xmm1.        |
| VEX.NDS.128.66.0F.WIG 12 /r<br>VMOVLPD xmm2, xmm1, m64 | RVM     | V/V                          | AVX                      | Merge double-precision floating-point value from m64 and the high quadword of xmm1. |
| EVEX.NDS.128.66.0F.W1 12 /r<br>VMOVLPD xmm2, xmm1, m64 | T1S     | V/V                          | AVX512F                  | Merge double-precision floating-point value from m64 and the high quadword of xmm1. |
| 66 0F 13/r<br>MOVLPD m64, xmm1                         | MR      | V/V                          | SSE2                     | Move double-precision floating-point value from low quadword of xmm1 to m64.        |
| VEX.128.66.0F.WIG 13/r<br>VMOVLPD m64, xmm1            | MR      | V/V                          | AVX                      | Move double-precision floating-point value from low quadword of xmm1 to m64.        |
| EVEX.128.66.0F.W1 13/r<br>VMOVLPD m64, xmm1            | T1S-MR  | V/V                          | AVX512F                  | Move double-precision floating-point value from low quadword of xmm1 to m64.        |

### 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:r/m (r)    | VEX.vvvv      | ModRM:r/m (r) | NA        |
| MR     | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |
| T1S    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |
| T1S-MR | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### **MOVLPD (128-bit Legacy SSE load)**

DEST[63:0] ← SRC[63:0]  
DEST[MAX\_VL-1:64] (Unmodified)

#### **VMOVLPD (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC2[63:0]  
DEST[127:64] ← SRC1[127:64]  
DEST[MAX\_VL-1:128] ← 0

#### **VMOVLPD (store)**

DEST[63:0] ← SRC[63:0]

### Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD \_\_m128d \_mm\_load\_pd ( \_\_m128d a, double \*p)  
MOVLPD void \_mm\_store\_pd (double \*p, \_\_m128d a)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD                      If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction                              | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| OF 12 /r<br>MOVLPS xmm1, m64                        | RM      | V/V                          | SSE                      | Move two packed single-precision floating-point values from m64 to low quadword of xmm1.        |
| VEX.NDS.128.OF.WIG 12 /r<br>VMOVLPS xmm2, xmm1, m64 | RVM     | V/V                          | AVX                      | Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1. |
| EVEX.NDS.128.OF.W0 12 /r<br>VMOVLPS xmm2, xmm1, m64 | T2      | V/V                          | AVX512F                  | Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1. |
| OF 13/r<br>MOVLPS m64, xmm1                         | MR      | V/V                          | SSE                      | Move two packed single-precision floating-point values from low quadword of xmm1 to m64.        |
| VEX.128.OF.WIG 13/r<br>VMOVLPS m64, xmm1            | MR      | V/V                          | AVX                      | Move two packed single-precision floating-point values from low quadword of xmm1 to m64.        |
| EVEX.128.OF.W0 13/r<br>VMOVLPS m64, xmm1            | T2-MR   | V/V                          | AVX512F                  | Move two packed single-precision floating-point values from low quadword of xmm1 to m64.        |

### 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      | ModRM:r/m (r) | NA        |
| MR    | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |
| T2    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |
| T2-MR | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.OF 13 /r) is legal and has the same behavior as the existing OF 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### **MOVLPS (128-bit Legacy SSE load)**

DEST[63:0] ← SRC[63:0]  
DEST[MAX\_VL-1:64] (Unmodified)

#### **VMOVLPS (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC2[63:0]  
DEST[127:64] ← SRC1[127:64]  
DEST[MAX\_VL-1:128] ← 0

#### **VMOVLPS (store)**

DEST[63:0] ← SRC[63:0]

### Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS \_\_m128 \_mm\_load\_pi ( \_\_m128 a, \_\_m64 \*p)  
MOVLPS void \_mm\_store\_pi (\_\_m64 \*p, \_\_m128 a)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD                    If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

| Opcode/<br>Instruction                            | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature Flag | Description   |
|---|------------|------------------------------|-----------------------|---|
| 66 0F 38 2A /r<br>MOVNTDQA xmm1, m128             | RM         | V/V                          | SSE4_1                | Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type. |
| VEX.128.66.0F38.WIG 2A /r<br>VMOVNTDQA xmm1, m128 | RM         | V/V                          | AVX                   | Move double quadword from m128 to xmm using non-temporal hint if WC memory type.  |
| VEX.256.66.0F38.WIG 2A /r<br>VMOVNTDQA ymm1, m256 | RM         | V/V                          | AVX2                  | Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.     |
| EVEX.128.66.0F38 2A /r<br>VMOVNTDQA xmm1, m128    | FVM        | V/V                          | AVX512VL<br>AVX512F   | Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.     |
| EVEX.256.66.0F38 2A /r<br>VMOVNTDQA ymm1, m256    | FVM        | V/V                          | AVX512VL<br>AVX512F   | Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.     |
| EVEX.512.66.0F38.W0 2A /r<br>VMOVNTDQA zmm1, m512 | FVM        | V/V                          | AVX512F               | Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.     |

### Instruction Operand Encoding<sup>1</sup>

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FVM   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and

1. ModRM.MOD = 011B required

writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

### Operation

#### MOVNTDQA (128bit- Legacy SSE form)

DEST ← SRC

DEST[MAX\_VL-1:128] (Unmodified)

#### VMOVNTDQA (VEX.128 and EVEX.128 encoded form)

DEST ← SRC

DEST[MAX\_VL-1:128] ← 0

#### VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

#### VMOVNTDQA (EVEX.512 encoded form)

DEST[511:0] ← SRC[511:0]

### Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA \_\_m512i \_mm512\_stream\_load\_si512(void \* p);

MOVNTDQA \_\_m128i \_mm\_stream\_load\_si128 (\_\_m128i \*p);

VMOVNTDQA \_\_m256i \_mm\_stream\_load\_si256 (\_\_m256i \*p);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 1;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

| Opcode/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| 66 0F E7 /r<br>MOVNTDQ m128, xmm1              | MR        | V/V                          | SSE2                  | Move packed integer values in xmm1 to m128 using non-temporal hint. |
| VEX.128.66.0F.WIG E7 /r<br>VMOVNTDQ m128, xmm1 | MR        | V/V                          | AVX                   | Move packed integer values in xmm1 to m128 using non-temporal hint. |
| VEX.256.66.0F.WIG E7 /r<br>VMOVNTDQ m256, ymm1 | MR        | V/V                          | AVX                   | Move packed integer values in ymm1 to m256 using non-temporal hint. |
| EVEX.128.66.0F E7 /r<br>VMOVNTDQ m128, xmm1    | FVM       | V/V                          | AVX512VL<br>AVX512F   | Move packed integer values in xmm1 to m128 using non-temporal hint. |
| EVEX.256.66.0F E7 /r<br>VMOVNTDQ m256, ymm1    | FVM       | V/V                          | AVX512VL<br>AVX512F   | Move packed integer values in zmm1 to m256 using non-temporal hint. |
| EVEX.512.66.0F.W0 E7 /r<br>VMOVNTDQ m512, zmm1 | FVM       | V/V                          | AVX512F               | Move packed integer values in zmm1 to m512 using non-temporal hint. |

### Instruction Operand Encoding<sup>1</sup>

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| MR    | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| FVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

1. ModRM.MOD = 011B required

### Operation

#### **VMOVNTDQ(EVEX encoded versions)**

VL = 128, 256, 512  
DEST[VL-1:0] ← SRC[VL-1:0]  
DEST[MAX\_VL-1:VL] ← 0

#### **MOVNTDQ (Legacy and VEX versions)**

DEST ← SRC

#### Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQ void \_mm512\_stream\_si512(void \* p, \_\_m512i a);  
VMOVNTDQ void \_mm256\_stream\_si256 (\_\_m256i \* p, \_\_m256i a);  
MOVNTDQ void \_mm\_stream\_si128 (\_\_m128i \* p, \_\_m128i a);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 1.SSE2;  
EVEX-encoded instruction, see Exceptions Type E1NF.  
#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.



## MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

| Opcode/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| 66 0F 2B /r<br>MOVNTPD m128, xmm1              | MR        | V/V                          | SSE2                     | Move packed double-precision values in xmm1 to m128 using non-temporal hint. |
| VEX.128.66.0F.WIG 2B /r<br>VMOVNTPD m128, xmm1 | MR        | V/V                          | AVX                      | Move packed double-precision values in xmm1 to m128 using non-temporal hint. |
| VEX.256.66.0F.WIG 2B /r<br>VMOVNTPD m256, ymm1 | MR        | V/V                          | AVX                      | Move packed double-precision values in ymm1 to m256 using non-temporal hint. |
| EVEX.128.66.0F.W1 2B /r<br>VMOVNTPD m128, xmm1 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move packed double-precision values in xmm1 to m128 using non-temporal hint. |
| EVEX.256.66.0F.W1 2B /r<br>VMOVNTPD m256, ymm1 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move packed double-precision values in ymm1 to m256 using non-temporal hint. |
| EVEX.512.66.0F.W1 2B /r<br>VMOVNTPD m512, zmm1 | FVM       | V/V                          | AVX512F                  | Move packed double-precision values in zmm1 to m512 using non-temporal hint. |

### Instruction Operand Encoding<sup>1</sup>

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| MR    | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| FVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

1. ModRM.MOD = 011B required

### Operation

#### **VMOVNTPD (EVEX encoded versions)**

VL = 128, 256, 512  
DEST[VL-1:0] ← SRC[VL-1:0]  
DEST[MAX\_VL-1:VL] ← 0

#### **MOVNTPD (Legacy and VEX versions)**

DEST ← SRC

#### Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void \_mm512\_stream\_pd(double \* p, \_\_m512d a);  
VMOVNTPD void \_mm256\_stream\_pd (double \* p, \_\_m256d a);  
MOVNTPD void \_mm\_stream\_pd (double \* p, \_\_m128d a);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 1.SSE2;  
EVEX-encoded instruction, see Exceptions Type E1NF.  
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

| Opcode/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| OF 2B /r<br>MOVNTPS m128, xmm1                 | MR        | V/V                          | SSE                      | Move packed single-precision values xmm1 to mem using non-temporal hint.     |
| VEX.128.OF.WIG 2B /r<br>VMOVNTPS m128, xmm1    | MR        | V/V                          | AVX                      | Move packed single-precision values xmm1 to mem using non-temporal hint.     |
| VEX.256.OF.WIG 2B /r<br>VMOVNTPS m256, ymm1    | MR        | V/V                          | AVX                      | Move packed single-precision values ymm1 to mem using non-temporal hint.     |
| EVEX.128.66.OF.W0 2B /r<br>VMOVNTPS m128, xmm1 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move packed single-precision values in xmm1 to m128 using non-temporal hint. |
| EVEX.256.66.OF.W0 2B /r<br>VMOVNTPS m256, ymm1 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move packed single-precision values in ymm1 to m256 using non-temporal hint. |
| EVEX.512.OF.W0 2B /r<br>VMOVNTPS m512, zmm1    | FVM       | V/V                          | AVX512F                  | Move packed single-precision values in zmm1 to m512 using non-temporal hint. |

### Instruction Operand Encoding<sup>1</sup>

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| MR    | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| FVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

1. ModRM.MOD = 011B required

### Operation

#### **VMOVNTPS (EVEX encoded versions)**

VL = 128, 256, 512  
DEST[VL-1:0] ← SRC[VL-1:0]  
DEST[MAX\_VL-1:VL] ← 0

#### **MOVNTPS**

DEST ← SRC

#### Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void \_\_mm512\_stream\_ps(float \* p, \_\_m512d a);  
MOVNTPS void \_\_mm\_stream\_ps (float \* p, \_\_m128d a);  
VMOVNTPS void \_\_mm256\_stream\_ps (float \* p, \_\_m256 a);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally  
EVEX-encoded instruction, see Exceptions Type E1NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction   | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|------------------------------|--------------------------|---|
| F2 0F 10 /r<br>MOVSD xmm1, xmm2                                | RM      | V/V                          | SSE2                     | Move scalar double-precision floating-point value from xmm2 to xmm1 register.                               |
| F2 0F 10 /r<br>MOVSD xmm1, m64                                 | RM      | V/V                          | SSE2                     | Load scalar double-precision floating-point value from m64 to xmm1 register.                                |
| F2 0F 11 /r<br>MOVSD xmm1/m64, xmm2                            | MR      | V/V                          | SSE2                     | Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.                           |
| VEX.NDS.LIG.F2.0F.WIG 10 /r<br>VMOVSD xmm1, xmm2, xmm3         | RVM     | V/V                          | AVX                      | Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register.                     |
| VEX.LIG.F2.0F.WIG 10 /r<br>VMOVSD xmm1, m64                    | XM      | V/V                          | AVX                      | Load scalar double-precision floating-point value from m64 to xmm1 register.                                |
| VEX.NDS.LIG.F2.0F.WIG 11 /r<br>VMOVSD xmm1, xmm2, xmm3         | MVR     | V/V                          | AVX                      | Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1.                    |
| VEX.LIG.F2.0F.WIG 11 /r<br>VMOVSD m64, xmm1                    | MR      | V/V                          | AVX                      | Store scalar double-precision floating-point value from xmm1 register to m64.                               |
| EVEX.NDS.LIG.F2.0F.W1 10 /r<br>VMOVSD xmm1 {k1}{z}, xmm2, xmm3 | RVM     | V/V                          | AVX512F                  | Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1. |
| EVEX.LIG.F2.0F.W1 10 /r<br>VMOVSD xmm1 {k1}{z}, m64            | T1S-RM  | V/V                          | AVX512F                  | Load scalar double-precision floating-point value from m64 to xmm1 register under writemask k1.             |
| EVEX.NDS.LIG.F2.0F.W1 11 /r<br>VMOVSD xmm1 {k1}{z}, xmm2, xmm3 | MVR     | V/V                          | AVX512F                  | Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1. |
| EVEX.LIG.F2.0F.W1 11 /r<br>VMOVSD m64 {k1}, xmm1               | T1S-MR  | V/V                          | AVX512F                  | Store scalar double-precision floating-point value from xmm1 register to m64 under writemask k1.            |

### 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        |
| MR     | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |
| XM     | ModRM:reg (w)    | ModRM:r/m (r) | NA            | NA        |
| MVR    | ModRM:r/m (w)    | vvvv (r)      | ModRM:reg (r) | NA        |
| T1S-RM | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| T1S-MR | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |

**Description**

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAX\_VL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAX\_VL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double-precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAX\_VL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

**Operation****VMOVSD (EVEX.NDS.LIG.F2.OF 10 /r: VMOVSD xmm1, m64 with support for 32 registers)**

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[511:64] ← 0

**VMOVSD (EVEX.NDS.LIG.F2.OF 11 /r: VMOVSD m64, xmm1 with support for 32 registers)**

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC[63:0]

ELSE \*DEST[63:0] remains unchanged\* ; merging-masking

FI;

**VMOVSD (EVEX.NDS.LIG.F2.OF 11 /r: VMOVSD xmm1, xmm2, xmm3)**

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC2[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)**

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] (Unmodified)

**VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ← SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ← SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)**

DEST[63:0] ← SRC[63:0]  
 DEST[MAX\_VL-1:64] ← 0

**MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)**

DEST[63:0] ← SRC[63:0]

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)**

DEST[63:0] ← SRC[63:0]  
 DEST[127:64] ← 0  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSD \_\_m128d \_\_mm\_mask\_load\_sd(\_\_m128d s, \_\_mmask8 k, double \* p);  
 VMOVSD \_\_m128d \_\_mm\_maskz\_load\_sd(\_\_mmask8 k, double \* p);  
 VMOVSD \_\_m128d \_\_mm\_mask\_move\_sd(\_\_m128d sh, \_\_mmask8 k, \_\_m128d sl, \_\_m128d a);  
 VMOVSD \_\_m128d \_\_mm\_maskz\_move\_sd(\_\_mmask8 k, \_\_m128d s, \_\_m128d a);  
 VMOVSD void \_\_mm\_mask\_store\_sd(double \* p, \_\_mmask8 k, \_\_m128d s);  
 MOVSD \_\_m128d \_\_mm\_load\_sd (double \*p)  
 MOVSD void \_\_mm\_store\_sd (double \*p, \_\_m128d a)  
 MOVSD \_\_m128d \_\_mm\_move\_sd ( \_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

**MOVSHDUP—Replicate Single FP Values**

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| F3 0F 16 /r<br>MOVSHDUP xmm1, xmm2/m128                         | RM        | V/V                          | SSE3                     | Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.                  |
| VEX.128.F3.0F.WIG 16 /r<br>VMOVSHDUP xmm1, xmm2/m128            | RM        | V/V                          | AVX                      | Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.                  |
| VEX.256.F3.0F.WIG 16 /r<br>VMOVSHDUP ymm1, ymm2/m256            | RM        | V/V                          | AVX                      | Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.                  |
| EVEX.128.F3.0F.W0 16 /r<br>VMOVSHDUP xmm1 {k1}{z},<br>xmm2/m128 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move odd index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask. |
| EVEX.256.F3.0F.W0 16 /r<br>VMOVSHDUP ymm1 {k1}{z},<br>ymm2/m256 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move odd index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask. |
| EVEX.512.F3.0F.W0 16 /r<br>VMOVSHDUP zmm1 {k1}{z},<br>zmm2/m512 | FVM       | V/V                          | AVX512F                  | Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FVM   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

**Description**

Duplicates odd-indexed single-precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 5-27. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.



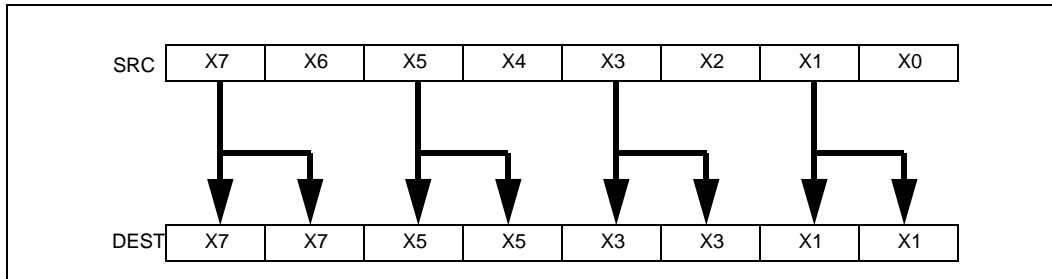


Figure 5-27. MOVSHDUP Operation

**Operation****VMOVSHDUP (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP\_SRC[31:0] ← SRC[63:32]

TMP\_SRC[63:32] ← SRC[63:32]

TMP\_SRC[95:64] ← SRC[127:96]

TMP\_SRC[127:96] ← SRC[127:96]

IF VL ≥ 256

    TMP\_SRC[159:128] ← SRC[191:160]

    TMP\_SRC[191:160] ← SRC[191:160]

    TMP\_SRC[223:192] ← SRC[255:224]

    TMP\_SRC[255:224] ← SRC[255:224]

FI;

IF VL ≥ 512

    TMP\_SRC[287:256] ← SRC[319:288]

    TMP\_SRC[319:288] ← SRC[319:288]

    TMP\_SRC[351:320] ← SRC[383:352]

    TMP\_SRC[383:352] ← SRC[383:352]

    TMP\_SRC[415:384] ← SRC[447:416]

    TMP\_SRC[447:416] ← SRC[447:416]

    TMP\_SRC[479:448] ← SRC[511:480]

    TMP\_SRC[511:480] ← SRC[511:480]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ← TMP\_SRC[i+31:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+31:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVSHDUP (VEX.256 encoded version)**

DEST[31:0] ← SRC[63:32]  
 DEST[63:32] ← SRC[63:32]  
 DEST[95:64] ← SRC[127:96]  
 DEST[127:96] ← SRC[127:96]  
 DEST[159:128] ← SRC[191:160]  
 DEST[191:160] ← SRC[191:160]  
 DEST[223:192] ← SRC[255:224]  
 DEST[255:224] ← SRC[255:224]  
 DEST[MAX\_VL-1:256] ← 0

**VMOVSHDUP (VEX.128 encoded version)**

DEST[31:0] ← SRC[63:32]  
 DEST[63:32] ← SRC[63:32]  
 DEST[95:64] ← SRC[127:96]  
 DEST[127:96] ← SRC[127:96]  
 DEST[MAX\_VL-1:128] ← 0

**MOVSHDUP (128-bit Legacy SSE version)**

DEST[31:0] ← SRC[63:32]  
 DEST[63:32] ← SRC[63:32]  
 DEST[95:64] ← SRC[127:96]  
 DEST[127:96] ← SRC[127:96]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSHDUP \_\_m512 \_\_mm512\_movehdup\_ps(\_\_m512 a);  
 VMOVSHDUP \_\_m512 \_\_mm512\_mask\_movehdup\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);  
 VMOVSHDUP \_\_m512 \_\_mm512\_maskz\_movehdup\_ps(\_\_mmask16 k, \_\_m512 a);  
 VMOVSHDUP \_\_m256 \_\_mm256\_mask\_movehdup\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);  
 VMOVSHDUP \_\_m256 \_\_mm256\_maskz\_movehdup\_ps(\_\_mmask8 k, \_\_m256 a);  
 VMOVSHDUP \_\_m128 \_\_mm\_mask\_movehdup\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);  
 VMOVSHDUP \_\_m128 \_\_mm\_maskz\_movehdup\_ps(\_\_mmask8 k, \_\_m128 a);  
 VMOVSHDUP \_\_m256 \_\_mm256\_movehdup\_ps(\_\_m256 a);  
 VMOVSHDUP \_\_m128 \_\_mm\_movehdup\_ps(\_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVSLDUP—Replicate Single FP Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| F3 0F 12 /r<br>MOVSLDUP xmm1, xmm2/m128                         | A         | V/V                          | SSE3                     | Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.                  |
| VEX.128.F3.0F.WIG 12 /r<br>VMOVSLDUP xmm1, xmm2/m128            | RM        | V/V                          | AVX                      | Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.                  |
| VEX.256.F3.0F.WIG 12 /r<br>VMOVSLDUP ymm1, ymm2/m256            | RM        | V/V                          | AVX                      | Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.                  |
| EVEX.128.F3.0F.W0 12 /r<br>VMOVSLDUP xmm1 {k1}{z},<br>xmm2/m128 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move even index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask. |
| EVEX.256.F3.0F.W0 12 /r<br>VMOVSLDUP ymm1 {k1}{z},<br>ymm2/m256 | FVM       | V/V                          | AVX512VL<br>AVX512F      | Move even index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask. |
| EVEX.512.F3.0F.W0 12 /r<br>VMOVSLDUP zmm1 {k1}{z},<br>zmm2/m512 | FVM       | V/V                          | AVX512F                  | Move even index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FVM   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 5-28. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

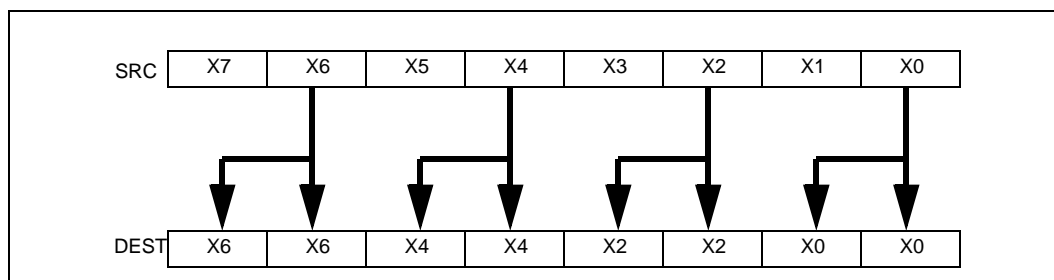


Figure 5-28. MOVSLDUP Operation

**Operation****VMOVSLDUP (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP\_SRC[31:0] ← SRC[31:0]

TMP\_SRC[63:32] ← SRC[31:0]

TMP\_SRC[95:64] ← SRC[95:64]

TMP\_SRC[127:96] ← SRC[95:64]

IF VL ≥ 256

TMP\_SRC[159:128] ← SRC[159:128]

TMP\_SRC[191:160] ← SRC[159:128]

TMP\_SRC[223:192] ← SRC[223:192]

TMP\_SRC[255:224] ← SRC[223:192]

FI;

IF VL ≥ 512

TMP\_SRC[287:256] ← SRC[287:256]

TMP\_SRC[319:288] ← SRC[287:256]

TMP\_SRC[351:320] ← SRC[351:320]

TMP\_SRC[383:352] ← SRC[351:320]

TMP\_SRC[415:384] ← SRC[415:384]

TMP\_SRC[447:416] ← SRC[415:384]

TMP\_SRC[479:448] ← SRC[479:448]

TMP\_SRC[511:480] ← SRC[479:448]

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVSLDUP (VEX.256 encoded version)**

DEST[31:0] ← SRC[31:0]

DEST[63:32] ← SRC[31:0]

DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]

DEST[159:128] ← SRC[159:128]

DEST[191:160] ← SRC[159:128]

DEST[223:192] ← SRC[223:192]

DEST[255:224] ← SRC[223:192]

DEST[MAX\_VL-1:256] ← 0

**VMOVSLDUP (VEX.128 encoded version)**

DEST[31:0] ← SRC[31:0]

DEST[63:32] ← SRC[31:0]

DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]

DEST[MAX\_VL-1:128] ← 0

**MOVSLDUP (128-bit Legacy SSE version)**

DEST[31:0] ← SRC[31:0]  
 DEST[63:32] ← SRC[31:0]  
 DEST[95:64] ← SRC[95:64]  
 DEST[127:96] ← SRC[95:64]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSLDUP \_\_m512 \_\_mm512\_moveldup\_ps( \_\_m512 a);  
 VMOVSLDUP \_\_m512 \_\_mm512\_mask\_moveldup\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);  
 VMOVSLDUP \_\_m512 \_\_mm512\_maskz\_moveldup\_ps( \_\_mmask16 k, \_\_m512 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_mask\_moveldup\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_maskz\_moveldup\_ps( \_\_mmask8 k, \_\_m256 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_mask\_moveldup\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_maskz\_moveldup\_ps( \_\_mmask8 k, \_\_m128 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_moveldup\_ps ( \_\_m256 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_moveldup\_ps ( \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD                    If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

**MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value**

| Opcode/<br>Instruction   | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|------------------------------|--------------------------|---|
| F3 0F 10 /r<br>MOVSS xmm1, xmm2                                | RM      | V/V                          | SSE                      | Merge scalar single-precision floating-point value from xmm2 to xmm1 register.                            |
| F3 0F 10 /r<br>MOVSS xmm1, m32                                 | RM      | V/V                          | SSE                      | Load scalar single-precision floating-point value from m32 to xmm1 register.                              |
| VEX.NDS.LIG.F3.0F.WIG 10 /r<br>VMOVSS xmm1, xmm2, xmm3         | RVM     | V/V                          | AVX                      | Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register                    |
| VEX.LIG.F3.0F.WIG 10 /r<br>VMOVSS xmm1, m32                    | XM      | V/V                          | AVX                      | Load scalar single-precision floating-point value from m32 to xmm1 register.                              |
| F3 0F 11 /r<br>MOVSS xmm2/m32, xmm1                            | MR      | V/V                          | SSE                      | Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.                         |
| VEX.NDS.LIG.F3.0F.WIG 11 /r<br>VMOVSS xmm1, xmm2, xmm3         | MVR     | V/V                          | AVX                      | Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.                    |
| VEX.LIG.F3.0F.WIG 11 /r<br>VMOVSS m32, xmm1                    | MR      | V/V                          | AVX                      | Move scalar single-precision floating-point value from xmm1 register to m32.                              |
| EVEX.NDS.LIG.F3.0F.W0 10 /r<br>VMOVSS xmm1 {k1}{z}, xmm2, xmm3 | RVM     | V/V                          | AVX512F                  | Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1. |
| EVEX.LIG.F3.0F.W0 10 /r<br>VMOVSS xmm1 {k1}{z}, m32            | T1S-RM  | V/V                          | AVX512F                  | Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.                   |
| EVEX.NDS.LIG.F3.0F.W0 11 /r<br>VMOVSS xmm1 {k1}{z}, xmm2, xmm3 | MVR     | V/V                          | AVX512F                  | Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1. |
| EVEX.LIG.F3.0F.W0 11 /r<br>VMOVSS m32 {k1}, xmm1               | T1S-MR  | V/V                          | AVX512F                  | Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.                   |

**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        |
| MR     | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |
| XM     | ModRM:reg (w)    | ModRM:r/m (r) | NA            | NA        |
| MVR    | ModRM:r/m (w)    | vvvv (r)      | ModRM:reg (r) | NA        |
| T1S-RM | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| T1S-MR | ModRM:r/m (w)    | ModRM:reg (r) | NA            | NA        |

## Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAX\_VL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAX\_VL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single-precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAX\_VL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

**VMOVSS (EVEX.NDS.LIG.F3.OF.WO 11 /r when the source operand is memory and the destination is an XMM register)**

```
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[511:32] ← 0
```

**VMOVSS (EVEX.NDS.LIG.F3.OF.WO 10 /r when the source operand is an XMM register and the destination is memory)**

```
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC[31:0]
    ELSE *DEST[31:0] remains unchanged* ; merging-masking
FI;
```

**VMOVSS (EVEX.NDS.LIG.F3.OF.WO 10/11 /r where the source and destination are XMM registers)**

```
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[127:32] ← SRC1[127:32]
```

DEST[MAX\_VL-1:128] ← 0

**MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)**

DEST[31:0] ← SRC[31:0]  
 DEST[MAX\_VL-1:32] (Unmodified)

**VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)**

DEST[31:0] ← SRC2[31:0]  
 DEST[127:32] ← SRC1[127:32]  
 DEST[MAX\_VL-1:128] ← 0

**VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)**

DEST[31:0] ← SRC2[31:0]  
 DEST[127:32] ← SRC1[127:32]  
 DEST[MAX\_VL-1:128] ← 0

**VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)**

DEST[31:0] ← SRC[31:0]  
 DEST[MAX\_VL-1:32] ← 0

**MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)**

DEST[31:0] ← SRC[31:0]

**MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)**

DEST[31:0] ← SRC[31:0]  
 DEST[127:32] ← 0  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSS \_\_m128 \_\_mm\_mask\_load\_ss(\_\_m128 s, \_\_mmask8 k, float \* p);  
 VMOVSS \_\_m128 \_\_mm\_maskz\_load\_ss(\_\_mmask8 k, float \* p);  
 VMOVSS \_\_m128 \_\_mm\_mask\_move\_ss(\_\_m128 sh, \_\_mmask8 k, \_\_m128 sl, \_\_m128 a);  
 VMOVSS \_\_m128 \_\_mm\_maskz\_move\_ss(\_\_mmask8 k, \_\_m128 s, \_\_m128 a);  
 VMOVSS void \_\_mm\_mask\_store\_ss(float \* p, \_\_mmask8 k, \_\_m128 a);  
 MOVSS \_\_m128 \_\_mm\_load\_ss(float \* p)  
 MOVSS void \_\_mm\_store\_ss(float \* p, \_\_m128 a)  
 MOVSS \_\_m128 \_\_mm\_move\_ss(\_\_m128 a, \_\_m128 b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.



## MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction                                     | Op / En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|-----------------------------|--------------------------|---|
| 66 0F 10 /r<br>MOVUPD xmm1, xmm2/m128                      | RM      | V/V                         | SSE2                     | Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.                            |
| 66 0F 11 /r<br>MOVUPD xmm2/m128, xmm1                      | MR      | V/V                         | SSE2                     | Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.                            |
| VEX.128.66.0F.WIG 10 /r<br>VMOVUPD xmm1, xmm2/m128         | RM      | V/V                         | AVX                      | Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.                            |
| VEX.128.66.0F.WIG 11 /r<br>VMOVUPD xmm2/m128, xmm1         | MR      | V/V                         | AVX                      | Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.                            |
| VEX.256.66.0F.WIG 10 /r<br>VMOVUPD ymm1, ymm2/m256         | RM      | V/V                         | AVX                      | Move unaligned packed double-precision floating-point from ymm2/mem to ymm1.                            |
| VEX.256.66.0F.WIG 11 /r<br>VMOVUPD ymm2/m256, ymm1         | MR      | V/V                         | AVX                      | Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem.                            |
| EVEX.128.66.0F.W1 10 /r<br>VMOVUPD xmm1 {k1}{z}, xmm2/m128 | FVM-RM  | V/V                         | AVX512VL<br>AVX512F      | Move unaligned packed double-precision floating-point from xmm2/m128 to xmm1 using writemask k1.        |
| EVEX.128.66.0F.W1 11 /r<br>VMOVUPD xmm2/m128 {k1}{z}, xmm1 | FVM-MR  | V/V                         | AVX512VL<br>AVX512F      | Move unaligned packed double-precision floating-point from xmm1 to xmm2/m128 using writemask k1.        |
| EVEX.256.66.0F.W1 10 /r<br>VMOVUPD ymm1 {k1}{z}, ymm2/m256 | FVM-RM  | V/V                         | AVX512VL<br>AVX512F      | Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1 using writemask k1.        |
| EVEX.256.66.0F.W1 11 /r<br>VMOVUPD ymm2/m256 {k1}{z}, ymm1 | FVM-MR  | V/V                         | AVX512VL<br>AVX512F      | Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256 using writemask k1.        |
| EVEX.512.66.0F.W1 10 /r<br>VMOVUPD zmm1 {k1}{z}, zmm2/m512 | FVM-RM  | V/V                         | AVX512F                  | Move unaligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |
| EVEX.512.66.0F.W1 11 /r<br>VMOVUPD zmm2/m512 {k1}{z}, zmm1 | FVM-MR  | V/V                         | AVX512F                  | Move unaligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1. |

### Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|--------|---------------|---------------|-----------|-----------|
| RM     | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| MR     | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| FVM-RM | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FVM-MR | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**EVEX.512 encoded version:**

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

**VEX.256 encoded version:**

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAX\_VL-1:256) of the destination register are zeroed.

**128-bit versions:**

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

**VEX.128 and EVEX.128 encoded versions:** Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation****VMOVUPD (EVEX encoded versions, register-copy form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ← SRC[i+63:i]

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

        ELSE DEST[i+63:i] ← 0 ; zeroing-masking

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVUPD (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ← SRC[i+63:i]

    ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

**VMOVUPD (EVEX encoded versions, load-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

VMOVUPD (VEX.256 encoded version, load - and register copy)

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVUPD (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVUPD (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**MOVUPD (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVUPD (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVUPD \_\_m512d \_\_mm512\_loadu\_pd( void \* s);

VMOVUPD \_\_m512d \_\_mm512\_mask\_loadu\_pd(\_\_m512d a, \_\_mmask8 k, void \* s);

VMOVUPD \_\_m512d \_\_mm512\_maskz\_loadu\_pd( \_\_mmask8 k, void \* s);

VMOVUPD void \_\_mm512\_storeu\_pd( void \* d, \_\_m512d a);

VMOVUPD void \_\_mm512\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VMOVUPD \_\_m256d \_\_mm256\_mask\_loadu\_pd(\_\_m256d s, \_\_mmask8 k, void \* m);

VMOVUPD \_\_m256d \_\_mm256\_maskz\_loadu\_pd( \_\_mmask8 k, void \* m);

VMOVUPD void \_\_mm256\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

VMOVUPD \_\_m128d \_\_mm\_mask\_loadu\_pd(\_\_m128d s, \_\_mmask8 k, void \* m);

VMOVUPD \_\_m128d \_\_mm\_maskz\_loadu\_pd( \_\_mmask8 k, void \* m);

VMOVUPD void \_\_mm\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m128d a);

MOVUPD \_\_m256d \_\_mm256\_loadu\_pd( double \* p);

MOVUPD void \_\_mm256\_storeu\_pd( double \*p, \_\_m256d a);

MOVUPD \_\_m128d \_\_mm\_loadu\_pd( double \* p);

MOVUPD void \_\_mm\_storeu\_pd( double \*p, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies; additionally

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction                                  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| OF 10 /r<br>MOVUPS xmm1, xmm2/m128                      | RM      | V/V                          | SSE                      | Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.                            |
| OF 11 /r<br>MOVUPS xmm2/m128, xmm1                      | MR      | V/V                          | SSE                      | Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.                            |
| VEX.128.OF.WIG 10 /r<br>VMOVUPS xmm1, xmm2/m128         | RM      | V/V                          | AVX                      | Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.                            |
| VEX.128.OF 11.WIG /r<br>VMOVUPS xmm2/m128, xmm1         | MR      | V/V                          | AVX                      | Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.                            |
| VEX.256.OF 10.WIG /r<br>VMOVUPS ymm1, ymm2/m256         | RM      | V/V                          | AVX                      | Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.                            |
| VEX.256.OF 11.WIG /r<br>VMOVUPS ymm2/m256, ymm1         | MR      | V/V                          | AVX                      | Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.                            |
| EVEX.128.OF.W0 10 /r<br>VMOVUPS xmm1 {k1}{z}, xmm2/m128 | FVM-RM  | V/V                          | AVX512VL<br>AVX512F      | Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.OF.W0 10 /r<br>VMOVUPS ymm1 {k1}{z}, ymm2/m256 | FVM-RM  | V/V                          | AVX512VL<br>AVX512F      | Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.OF.W0 10 /r<br>VMOVUPS zmm1 {k1}{z}, zmm2/m512 | FVM-RM  | V/V                          | AVX512F                  | Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |
| EVEX.128.OF.W0 11 /r<br>VMOVUPS xmm2/m128 {k1}{z}, xmm1 | FVM-MR  | V/V                          | AVX512VL<br>AVX512F      | Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1. |
| EVEX.256.OF.W0 11 /r<br>VMOVUPS ymm2/m256 {k1}{z}, ymm1 | FVM-MR  | V/V                          | AVX512VL<br>AVX512F      | Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1. |
| EVEX.512.OF.W0 11 /r<br>VMOVUPS zmm2/m512 {k1}{z}, zmm1 | FVM-MR  | V/V                          | AVX512F                  | Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1. |

### Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|--------|---------------|---------------|-----------|-----------|
| RM     | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| MR     | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |
| FVM-RM | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FVM-MR | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**EVEX.512 encoded version:**

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

**VEX.256 and EVEX.256 encoded versions:**

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAX\_VL-1:256) of the destination register are zeroed.

**128-bit versions:**

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

**VEX.128 and EVEX.128 encoded versions:** Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation****VMOVUPS (EVEX encoded versions, register-copy form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[i+31:i]

  ELSE

    IF \*merging-masking\* ;merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE DEST[i+31:i] ← 0 ;zeroing-masking

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVUPS (EVEX encoded versions, store-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[i+31:i]

  ELSE \*DEST[i+31:i] remains unchanged\* ;merging-masking

  FI;

ENDFOR;

**VMOVUPS (EVEX encoded versions, load-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVUPS (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVUPS (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVUPS (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**MOVUPS (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVUPS (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVUPS \_\_m512 \_\_mm512\_loadu\_ps( void \* s);

VMOVUPS \_\_m512 \_\_mm512\_mask\_loadu\_ps(\_\_m512 a, \_\_mmask16 k, void \* s);

VMOVUPS \_\_m512 \_\_mm512\_maskz\_loadu\_ps( \_\_mmask16 k, void \* s);

VMOVUPS void \_\_mm512\_storeu\_ps( void \* d, \_\_m512 a);

VMOVUPS void \_\_mm512\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m512 a);

VMOVUPS \_\_m256 \_\_mm256\_mask\_loadu\_ps(\_\_m256 a, \_\_mmask8 k, void \* s);

VMOVUPS \_\_m256 \_\_mm256\_maskz\_loadu\_ps( \_\_mmask8 k, void \* s);

VMOVUPS void \_\_mm256\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VMOVUPS \_\_m128 \_\_mm\_mask\_loadu\_ps(\_\_m128 a, \_\_mmask8 k, void \* s);

VMOVUPS \_\_m128 \_\_mm\_maskz\_loadu\_ps( \_\_mmask8 k, void \* s);

VMOVUPS void \_\_mm\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

MOVUPS \_\_m256 \_\_mm256\_loadu\_ps ( float \* p);

MOVUPS void \_\_mm256\_storeu\_ps( float \*p, \_\_m256 a);

MOVUPS \_\_m128 \_\_mm\_loadu\_ps ( float \* p);

MOVUPS void \_\_mm\_storeu\_ps( float \*p, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## PSADBW—Compute Sum of Absolute Differences

| Opcode/<br>Instruction   | Op /<br>En | Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|-----------------|--------------------------|--|
| 66 0F F6 /r<br>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 F6 /r<br>VPSADBW xmm1, xmm2,<br>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 F6 /r<br>VPSADBW ymm1, ymm2,<br>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.              |
| EVEX.NDS.128.66.0F.WIG F6 /r<br>VPSADBW xmm1, xmm2,<br>xmm3/m128 | FVM        | V/V             | AVX512VL<br>AVX512BW     | Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.              |
| EVEX.NDS.256.66.0F.WIG F6 /r<br>VPSADBW ymm1, ymm2,<br>ymm3/m256 | FVM        | V/V             | AVX512VL<br>AVX512BW     | Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.              |
| EVEX.NDS.512.66.0F.WIG F6 /r<br>VPSADBW zmm1, zmm2,<br>zmm3/m512 | FVM        | V/V             | AVX512BW                 | Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.              |

### 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      | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Computes the absolute value of the difference of packed groups of 8 unsigned byte integers from the second source operand and from the first source operand. The first 8 differences are summed to produce an unsigned word integer that is stored in the low word of the destination; the second 8 differences are summed to produce an unsigned word in bits 79:64 of the destination.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

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 (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

### Operation

#### VPSADBW (EVEX encoded versions)

VL = 128, 256, 512

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 1 through 15 \*)

TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 00000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)

DEST[127:80] ← 00000000000000H

IF VL >= 256

(\* Repeat operation for bytes 16 through 31 \*)

TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])

DEST[143:128] ← SUM(TEMP16:TEMP23)

DEST[191:144] ← 00000000000000H

DEST[207:192] ← SUM(TEMP24:TEMP31)

DEST[223:208] ← 00000000000000H

FI;

IF VL >= 512

(\* Repeat operation for bytes 32 through 63 \*)

TEMP63 ← ABS(SRC1[511:504] - SRC2[511:504])

DEST[271:256] ← SUM(TEMP0:TEMP7)

DEST[319:272] ← 00000000000000H

DEST[335:320] ← SUM(TEMP8:TEMP15)

DEST[383:336] ← 00000000000000H

DEST[399:384] ← SUM(TEMP16:TEMP23)

DEST[447:400] ← 00000000000000H

DEST[463:448] ← SUM(TEMP24:TEMP31)

DEST[511:464] ← 00000000000000H

FI;

DEST[MAX\_VL-1:VL] ← 0

#### VPSADBW (VEX.256 encoded version)

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 2 through 30 \*)

TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 00000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)

DEST[127:80] ← 00000000000000H

DEST[143:128] ← SUM(TEMP16:TEMP23)

DEST[191:144] ← 00000000000000H

DEST[207:192] ← SUM(TEMP24:TEMP31)

DEST[223:208] ← 00000000000000H

DEST[MAX\_VL-1:256] ← 0

**VPSADBW (VEX.128 encoded version)**

TEMP0  $\leftarrow$  ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 2 through 14 \*)

TEMP15  $\leftarrow$  ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)

DEST[63:16]  $\leftarrow$  000000000000H

DEST[79:64]  $\leftarrow$  SUM(TEMP8:TEMP15)

DEST[127:80]  $\leftarrow$  000000000000H

DEST[MAX\_VL-1:128]  $\leftarrow$  0

**PSADBW (128-bit Legacy SSE version)**

TEMP0  $\leftarrow$  ABS(DEST[7:0] - SRC[7:0])

(\* Repeat operation for bytes 2 through 14 \*)

TEMP15  $\leftarrow$  ABS(DEST[127:120] - SRC[127:120])

DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)

DEST[63:16]  $\leftarrow$  000000000000H

DEST[79:64]  $\leftarrow$  SUM(TEMP8:TEMP15)

DEST[127:80]  $\leftarrow$  000000000000

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSADBW \_\_m512i\_mm512\_sad\_epu8(\_\_m512i a, \_\_m512i b)

(V)PSADBW \_\_m128i\_mm\_sad\_epu8(\_\_m128i a, \_\_m128i b)

VPSADBW \_\_m256i\_mm256\_sad\_epu8(\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

**MULPD—Multiply Packed Double-Precision Floating-Point Values**

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F 59 /r<br>MULPD xmm1, xmm2/m128   | RM        | V/V                          | SSE2                     | Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.         |
| VEX.NDS.128.66.0F.WIG 59 /r<br>VMULPD xmm1, xmm2, xmm3/m128                        | RVM       | V/V                          | AVX                      | Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.         |
| VEX.NDS.256.66.0F.WIG 59 /r<br>VMULPD ymm1, ymm2, ymm3/m256                        | RVM       | V/V                          | AVX                      | Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.         |
| EVEX.NDS.128.66.0F.W1 59 /r<br>VMULPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1. |
| EVEX.NDS.256.66.0F.W1 59 /r<br>VMULPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1. |
| EVEX.NDS.512.66.0F.W1 59 /r<br>VMULPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1. |

**Instruction Operand Encoding**

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Multiply packed double-precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VMULPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMULPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] \* SRC2[63:0]

DEST[127:64] ← SRC1[127:64] \* SRC2[127:64]

DEST[191:128] ← SRC1[191:128] \* SRC2[191:128]

DEST[255:192] ← SRC1[255:192] \* SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0;

**VMULPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] \* SRC2[63:0]

DEST[127:64] ← SRC1[127:64] \* SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

**MULPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] \* SRC[63:0]

DEST[127:64] ← DEST[127:64] \* SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMULPD \_\_m512d \_\_mm512\_mul\_pd( \_\_m512d a, \_\_m512d b);

VMULPD \_\_m512d \_\_mm512\_mask\_mul\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VMULPD \_\_m512d \_\_mm512\_maskz\_mul\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VMULPD \_\_m512d \_\_mm512\_mul\_round\_pd( \_\_m512d a, \_\_m512d b, int);

VMULPD \_\_m512d \_\_mm512\_mask\_mul\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMULPD \_\_m512d \_\_mm512\_maskz\_mul\_round\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMULPD \_\_m256d \_\_mm256\_mul\_pd( \_\_m256d a, \_\_m256d b);

MULPD \_\_m128d \_mm\_mul\_pd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MULPS—Multiply Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| OF 59 /r<br>MULPS xmm1, xmm2/m128  | RM         | V/V                          | SSE                      | Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.         |
| VEX.NDS.128.OF.WIG 59 /r<br>VMULPS xmm1, xmm2, xmm3/m128                         | RVM        | V/V                          | AVX                      | Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.         |
| VEX.NDS.256.OF.WIG 59 /r<br>VMULPS ymm1, ymm2, ymm3/m256                         | RVM        | V/V                          | AVX                      | Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.         |
| EVEX.NDS.128.OF.WO 59 /r<br>VMULPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1. |
| EVEX.NDS.256.OF.WO 59 /r<br>VMULPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1. |
| EVEX.NDS.512.OF.WO 59 /r<br>VMULPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er} | FV         | V/V                          | AVX512F                  | Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VMULPS (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMULPS (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[159:128] ← SRC1[159:128] * SRC2[159:128]
DEST[191:160] ← SRC1[191:160] * SRC2[191:160]
DEST[223:192] ← SRC1[223:192] * SRC2[223:192]
DEST[255:224] ← SRC1[255:224] * SRC2[255:224].
DEST[MAX_VL-1:256] ← 0;

```

**VMULPS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**MULPS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```



**Intel C/C++ Compiler Intrinsic Equivalent**

VMULPS \_\_m512 \_\_mm512\_mul\_ps( \_\_m512 a, \_\_m512 b);  
 VMULPS \_\_m512 \_\_mm512\_mask\_mul\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMULPS \_\_m512 \_\_mm512\_maskz\_mul\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMULPS \_\_m512 \_\_mm512\_mul\_round\_ps( \_\_m512 a, \_\_m512 b, int);  
 VMULPS \_\_m512 \_\_mm512\_mask\_mul\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMULPS \_\_m512 \_\_mm512\_maskz\_mul\_round\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMULPS \_\_m256 \_\_mm256\_mask\_mul\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMULPS \_\_m256 \_\_mm256\_maskz\_mul\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMULPS \_\_m128 \_\_mm\_mask\_mul\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMULPS \_\_m128 \_\_mm\_maskz\_mul\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMULPS \_\_m256 \_\_mm256\_mul\_ps( \_\_m256 a, \_\_m256 b);  
 MULPS \_\_m128 \_\_mm\_mul\_ps( \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MULSD—Multiply Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| F2 0F 59 /r<br>MULSD xmm1,xmm2/m64   | RM         | V/V                          | SSE2                     | Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1. |
| VEX.NDS.128.F2.0F.WIG 59 /r<br>VMULSD xmm1,xmm2, xmm3/m64                  | RVM        | V/V                          | AVX                      | Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2. |
| EVEX.NDS.LIG.F2.0F.W1 59 /r<br>VMULSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64 {er} | T1S        | V/V                          | AVX512F                  | Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2. |

### Instruction Operand Encoding

| Op/En | 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        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VMULSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN  DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI
  FI;
ENDIFOR
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VMULSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**MULSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULSD __m128d __mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d __mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d __mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d __mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d __mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d __mm_mul_sd( __m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## MULSS—Multiply Scalar Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| F3 0F 59 /r<br>MULSS xmm1,xmm2/m32   | RM         | V/V                          | SSE                      | Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1. |
| VEX.NDS.128.F3.0F.WIG 59 /r<br>VMULSS xmm1,xmm2, xmm3/m32                  | RVM        | V/V                          | AVX                      | Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2. |
| EVEX.NDS.LIG.F3.0F.W0 59 /r<br>VMULSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32 {er} | T1S        | V/V                          | AVX512F                  | Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2. |

### Instruction Operand Encoding

| Op/En | 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        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VMULSS (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI
  FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMULSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**MULSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] * SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULSS __m128 __mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 __mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 __mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 __mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 __mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 __mm_mul_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Underflow, Overflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 56/r<br>ORPD xmm1, xmm2/m128  | RM         | V/V                          | SSE2                     | Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem                                   |
| VEX.NDS.128.66.0F 56 /r<br>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 56 /r<br>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                                   |
| EVEX.NDS.128.66.0F.W1 56 /r<br>VORPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1. |
| EVEX.NDS.256.66.0F.W1 56 /r<br>VORPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1. |
| EVEX.NDS.512.66.0F.W1 56 /r<br>VORPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512DQ                 | Return the bitwise logical OR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a bitwise logical OR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

### Operation

#### VORPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[63:0]
        ELSE
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VORPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAX_VL-1:256] ← 0

```

**VORPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAX_VL-1:128] ← 0

```

**ORPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VORPD __m512d __mm512_or_pd (__m512d a, __m512d b);
VORPD __m512d __mm512_mask_or_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d __mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);
VORPD __m256d __mm256_mask_or_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VORPD __m256d __mm256_maskz_or_pd (__mmask8 k, __m256d a, __m256d b);
VORPD __m128d __mm_mask_or_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d __mm_maskz_or_pd (__mmask8 k, __m128d a, __m128d b);
VORPD __m256d __mm256_or_pd (__m256d a, __m256d b);
ORPD __m128d __mm_or_pd (__m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| OF 56 /r<br>ORPS xmm1, xmm2/m128   | RM         | V/V                          | SSE                      | Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem                                   |
| VEX.NDS.128.OF 56 /r<br>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.OF 56 /r<br>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                                   |
| EVEX.NDS.128.OF.W0 56 /r<br>VORPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1. |
| EVEX.NDS.256.OF.W0 56 /r<br>VORPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1. |
| EVEX.NDS.512.OF.W0 56 /r<br>VORPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512DQ                 | Return the bitwise logical OR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a bitwise logical OR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

### Operation

#### VORPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
        ELSE
          DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VORPS (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAX_VL-1:256] ← 0

```

**VORPS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**ORPS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VORPS __m512 __mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 __mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 __mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 __mm256_mask_or_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VORPS __m256 __mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 __mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 __mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 __mm256_or_ps (__m256 a, __m256 b);

```

ORPS \_\_m128 \_\_mm\_or\_ps (\_\_m128 a, \_\_m128 b);

#### **SIMD Floating-Point Exceptions**

None

#### **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**PABSBB/PABSBB/PABSBB/PABSBB—Packed Absolute Value**

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature Flag | Description  |
|--|------------|------------------------------|-----------------------|--|
| 66 0F 38 1C /r<br>PABSBB xmm1, xmm2/m128                                 | RM         | V/V                          | SSSE3                 | Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.                                      |
| 66 0F 38 1D /r<br>PABSBB xmm1, xmm2/m128                                 | RM         | V/V                          | SSSE3                 | Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.                            |
| 66 0F 38 1E /r<br>PABSBB 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<br>VPABSBB 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<br>VPABSBB 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<br>VPABSBB 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<br>VPABSBB 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<br>VPABSBB 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<br>VPABSBB ymm1, ymm2/m256                     | RM         | V/V                          | AVX2                  | Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.                            |
| EVEX.128.66.0F 38 1C /r<br>VPABSBB xmm1 {k1}{z}, xmm2/m128               | FVM        | V/V                          | AVX512VL<br>AVX512BW  | Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.                   |
| EVEX.256.66.0F 38 1C /r<br>VPABSBB ymm1 {k1}{z}, ymm2/m256               | FVM        | V/V                          | AVX512VL<br>AVX512BW  | Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.                   |
| EVEX.512.66.0F 38 1C /r<br>VPABSBB zmm1 {k1}{z}, zmm2/m512               | FVM        | V/V                          | AVX512BW              | Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.                   |
| EVEX.128.66.0F 38 1D /r<br>VPABSBB xmm1 {k1}{z}, xmm2/m128               | FVM        | V/V                          | AVX512VL<br>AVX512BW  | Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.         |
| EVEX.256.66.0F 38 1D /r<br>VPABSBB ymm1 {k1}{z}, ymm2/m256               | FVM        | V/V                          | AVX512VL<br>AVX512BW  | Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.         |
| EVEX.512.66.0F 38 1D /r<br>VPABSBB zmm1 {k1}{z}, zmm2/m512               | FVM        | V/V                          | AVX512BW              | Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.         |
| EVEX.128.66.0F.W0 38 1E /r<br>VPABSBB xmm1 {k1}{z},<br>xmm2/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F   | Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1. |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature Flag | Description  |
|---|------------|------------------------------|-----------------------|--|
| EVEX.256.66.0F.W0 38 1E /r<br>VPABSD ymm1 {k1}{z},<br>ymm2/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F   | Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 1E /r<br>VPABSD zmm1 {k1}{z},<br>zmm2/m512/m32bcst  | FV         | V/V                          | AVX512F               | Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1. |
| EVEX.128.66.0F.W1 38 1F /r<br>VPABSQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F   | Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1. |
| EVEX.256.66.0F.W1 38 1F /r<br>VPABSQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F   | Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 1F /r<br>VPABSQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst  | FV         | V/V                          | AVX512F               | Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FVM   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL\_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### PABSB with 128 bit operands:

Unsigned DEST[7:0] ← ABS(SRC[7: 0])

Repeat operation for 2nd through 15th bytes  
 Unsigned DEST[127:120] ← ABS(SRC[127:120])

**VPABS B 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])

**VPABS B with 256 bit operands:**

Unsigned DEST[7:0] ← ABS(SRC[7:0])  
 Repeat operation for 2nd through 31st bytes  
 Unsigned DEST[255:248] ← ABS(SRC[255:248])

**VPABS B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        DEST[i+7:i] ← 0
      FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**PABS W 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])

**VPABS W 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])

**VPABS W with 256 bit operands:**

Unsigned DEST[15:0] ← ABS(SRC[15:0])  
 Repeat operation for 2nd through 15th 16-bit words  
 Unsigned DEST[255:240] ← ABS(SRC[255:240])

**VPABS W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[i+15:i] ← ABS(SRC[i+15:i])
    ELSE

```

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+15:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[j+15:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**PABSD with 128 bit operands:**

```

    Unsigned DEST[31:0] ← ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] ← ABS(SRC[127:96])

```

**VPABSD with 128 bit operands:**

```

    Unsigned DEST[31:0] ← ABS(SRC[31:0])
    Repeat operation for 2nd through 3rd 32-bit double words
    Unsigned DEST[127:96] ← ABS(SRC[127:96])

```

**VPABSD with 256 bit operands:**

```

    Unsigned DEST[31:0] ← ABS(SRC[31:0])
    Repeat operation for 2nd through 7th 32-bit double words
    Unsigned DEST[255:224] ← ABS(SRC[255:224])

```

**VPABSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[j+31:i] ← ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[j+31:i] ← ABS(SRC[j+31:i])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[j+31:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[j+31:i] ← 0
            FI
        FI;
    ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPABSQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN

```

```

        Unsigned DEST[i+63:i] ← ABS(SRC[63:0])
    ELSE
        Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])
    FI;
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPABSB__m512i__mm512_abs_epi8 (__m512i a)
VPABSW__m512i__mm512_abs_epi16 (__m512i a)
VPABSB__m512i__mm512_mask_abs_epi8 (__m512i s, __mmask64 m, __m512i a)
VPABSW__m512i__mm512_mask_abs_epi16 (__m512i s, __mmask32 m, __m512i a)
VPABSB__m512i__mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW__m512i__mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB__m256i__mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW__m256i__mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB__m256i__mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW__m256i__mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB__m128i__mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW__m128i__mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB__m128i__mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW__m128i__mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i__mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i__mm256_maskz_abs_epi32( __mmask8 k, __m256i a);
VPABSD __m128i__mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i__mm_maskz_abs_epi32( __mmask8 k, __m128i a);
VPABSD __m512i__mm512_abs_epi32( __m512i a);
VPABSD __m512i__mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD __m512i__mm512_maskz_abs_epi32( __mmask16 k, __m512i a);
VPABSQ __m512i__mm512_abs_epi64( __m512i a);
VPABSQ __m512i__mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i__mm512_maskz_abs_epi64( __mmask8 k, __m512i a);
VPABSQ __m256i__mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i__mm256_maskz_abs_epi64( __mmask8 k, __m256i a);
VPABSQ __m128i__mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i__mm_maskz_abs_epi64( __mmask8 k, __m128i a);
PABSB __m128i__mm_abs_epi8 (__m128i a)
VPABSB __m128i__mm_abs_epi8 (__m128i a)
VPABSB __m256i__mm256_abs_epi8 (__m256i a)
PABSW __m128i__mm_abs_epi16 (__m128i a)
VPABSW __m128i__mm_abs_epi16 (__m128i a)
VPABSW __m256i__mm256_abs_epi16 (__m256i a)
PABSD __m128i__mm_abs_epi32 (__m128i a)
VPABSD __m128i__mm_abs_epi32 (__m128i a)
VPABSD __m256i__mm256_abs_epi32 (__m256i a)

```



**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPABSD/Q, see Exceptions Type E4.

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.

## PACKSSWB/PACKSSDW—Pack with Signed Saturation

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 63 /r<br>PACKSSWB <i>xmm1, xmm2/m128</i>  | RM         | V/V                          | SSE2                     | Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.                                     |
| 66 0F 6B /r<br>PACKSSDW <i>xmm1, xmm2/m128</i>  | RM         | V/V                          | SSE2                     | Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.                                |
| VEX.NDS.128.66.0F 63 /r<br>VPACKSSWB <i>xmm1, xmm2, xmm3/m128</i>                     | RVM        | V/V                          | AVX                      | Converts 8 packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.                                     |
| VEX.NDS.128.66.0F 6B /r<br>VPACKSSDW <i>xmm1, xmm2, xmm3/m128</i>                     | RVM        | V/V                          | AVX                      | Converts 4 packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.                                |
| VEX.NDS.256.66.0F 63 /r<br>VPACKSSWB <i>ymm1, ymm2, ymm3/m256</i>                     | RVM        | V/V                          | AVX2                     | Converts 16 packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 32 packed signed byte integers in <i>ymm1</i> using signed saturation.                                    |
| VEX.NDS.256.66.0F 6B /r<br>VPACKSSDW <i>ymm1, ymm2, ymm3/m256</i>                     | RVM        | V/V                          | AVX2                     | Converts 8 packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 16 packed signed word integers in <i>ymm1</i> using signed saturation.                               |
| EVEX.NDS.128.66.0F.WIG 63 /r<br>VPACKSSWB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into packed signed byte integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .               |
| EVEX.NDS.256.66.0F.WIG 63 /r<br>VPACKSSWB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into packed signed byte integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .               |
| EVEX.NDS.512.66.0F.WIG 63 /r<br>VPACKSSWB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>        | FVM        | V/V                          | AVX512BW                 | Converts packed signed word integers from <i>zmm2</i> and from <i>zmm3/m512</i> into packed signed byte integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .               |
| EVEX.NDS.128.66.0F.W0 6B /r<br>VPACKSSDW <i>xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</i> | FV         | V/V                          | AVX512VL<br>AVX512BW     | Converts packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128/m32bcst</i> into packed signed word integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.W0 6B /r<br>VPACKSSDW <i>ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</i> | FV         | V/V                          | AVX512VL<br>AVX512BW     | Converts packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256/m32bcst</i> into packed signed word integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.W0 6B /r<br>VPACKSSDW <i>zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</i> | FV         | V/V                          | AVX512BW                 | Converts packed signed doubleword integers from <i>zmm2</i> and from <i>zmm3/m512/m32bcst</i> into packed signed word integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> . |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed doubleword value is beyond the range of an unsigned word (i.e. greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM destination register destination are unmodified.

## Operation

## PACKSSWB instruction (128-bit Legacy SSE version)

```

DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAX_VL-1:128] (Unmodified)

```

**PACKSSDW instruction (128-bit Legacy SSE version)**

DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);  
 DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);  
 DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);  
 DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);  
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);  
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);  
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);  
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPACKSSWB instruction (VEX.128 encoded version)**

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 DEST[MAX\_VL-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[MAX\_VL-1:128] ← 0;

**VPACKSSWB instruction (VEX.256 encoded version)**

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);

DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);  
 DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);  
 DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);  
 DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);  
 DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);  
 DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);  
 DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);  
 DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);  
 DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);  
 DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);  
 DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);  
 DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);  
 DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);  
 DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);  
 DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);  
 DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);  
 DEST[MAX\_VL-1:256] ← 0;

#### VPACKSSDW instruction (VEX.256 encoded version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);  
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);  
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);  
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);  
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);  
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);  
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);  
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);  
 DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);  
 DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);  
 DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);  
 DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);  
 DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);  
 DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);  
 DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);  
 DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);  
 DEST[MAX\_VL-1:256] ← 0;

#### VPACKSSWB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

TMP\_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 TMP\_DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 TMP\_DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 TMP\_DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 TMP\_DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 TMP\_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 TMP\_DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 TMP\_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 TMP\_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 TMP\_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 TMP\_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);

```

TMP_DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToSignedByte (SRC2[431:416]);

```

```

    TMP_DEST[479:472] ← SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] ← SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

VPACKSSDW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;

TMP_DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] ← SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] ← SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] ← SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] ← SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] ← SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] ← SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] ← SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] ← SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] ← SaturateSignedDwordToSignedWord (SRC1[287:256]);

```

```

TMP_DEST[287:272] ← SaturateSignedDwordToSignedWord (SRC1[319:288]);
TMP_DEST[303:288] ← SaturateSignedDwordToSignedWord (SRC1[351:320]);
TMP_DEST[319:304] ← SaturateSignedDwordToSignedWord (SRC1[383:352]);
TMP_DEST[335:320] ← SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
TMP_DEST[351:336] ← SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
TMP_DEST[367:352] ← SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
TMP_DEST[383:368] ← SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

TMP_DEST[399:384] ← SaturateSignedDwordToSignedWord (SRC1[415:384]);
TMP_DEST[415:400] ← SaturateSignedDwordToSignedWord (SRC1[447:416]);
TMP_DEST[431:416] ← SaturateSignedDwordToSignedWord (SRC1[479:448]);
TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
TMP_DEST[479:464] ← SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);

FI;
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[j+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[j+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKSSDW __m512i __mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW __m512i __mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW __m512i __mm512_maskz_packs_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW __m256i __mm256_mask_packs_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW __m256i __mm256_maskz_packs_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW __m128i __mm128_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW __m128i __mm128_maskz_packs_epi32(__mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m512i __mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB __m512i __mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB __m512i __mm512_maskz_packs_epi16(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB __m256i __mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB __m256i __mm256_maskz_packs_epi16(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB __m128i __mm128_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m128i __mm128_maskz_packs_epi16(__mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i __mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i __mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i __mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i __mm256_packs_epi32(__m256i m1, __m256i m2)

```

### SIMD Floating-Point Exceptions

None



**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.

## PACKUSDW—Pack with Unsigned Saturation

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F 38 2B /r<br>PACKUSDW <i>xmm1, xmm2/m128</i>                                      | RM         | V/V                          | SSE4_1                   | Convert 4 packed signed doubleword integers from <i>xmm1</i> and 4 packed signed doubleword integers from <i>xmm2/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.                              |
| VEX.NDS.128.66.0F38 2B /r<br>VPACKUSDW <i>xmm1, xmm2, xmm3/m128</i>                    | RVM        | V/V                          | AVX                      | Convert 4 packed signed doubleword integers from <i>xmm2</i> and 4 packed signed doubleword integers from <i>xmm3/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.                              |
| VEX.NDS.256.66.0F38 2B /r<br>VPACKUSDW <i>ymm1, ymm2, ymm3/m256</i>                    | RVM        | V/V                          | AVX2                     | Convert 8 packed signed doubleword integers from <i>ymm2</i> and 8 packed signed doubleword integers from <i>ymm3/m256</i> into 16 packed unsigned word integers in <i>ymm1</i> using unsigned saturation.                             |
| EVEX.NDS.128.66.0F38.W0 2B /r<br>VPACKUSDW <i>xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</i> | FV         | V/V                          | AVX512VL<br>AVX512BW     | Convert packed signed doubleword integers from <i>xmm2</i> and packed signed doubleword integers from <i>xmm3/m128/m32bcst</i> into packed unsigned word integers in <i>xmm1</i> using unsigned saturation under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F38.W0 2B /r<br>VPACKUSDW <i>ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst</i> | FV         | V/V                          | AVX512VL<br>AVX512BW     | Convert packed signed doubleword integers from <i>ymm2</i> and packed signed doubleword integers from <i>ymm3/m256/m32bcst</i> into packed unsigned word integers in <i>ymm1</i> using unsigned saturation under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F38.W0 2B /r<br>VPACKUSDW <i>zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst</i> | FV         | V/V                          | AVX512BW                 | Convert packed signed doubleword integers from <i>zmm2</i> and packed signed doubleword integers from <i>zmm3/m512/m32bcst</i> into packed unsigned word integers in <i>zmm1</i> using unsigned saturation under writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

#### Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask *k1*.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding destination register destination are unmodified.

**Operation****PACKUSDW (Legacy SSE instruction)**

TMP[15:0]  $\leftarrow$  (DEST[31:0] < 0) ? 0 : DEST[15:0];  
 DEST[15:0]  $\leftarrow$  (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0];  
 TMP[31:16]  $\leftarrow$  (DEST[63:32] < 0) ? 0 : DEST[47:32];  
 DEST[31:16]  $\leftarrow$  (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16];  
 TMP[47:32]  $\leftarrow$  (DEST[95:64] < 0) ? 0 : DEST[79:64];  
 DEST[47:32]  $\leftarrow$  (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32];  
 TMP[63:48]  $\leftarrow$  (DEST[127:96] < 0) ? 0 : DEST[111:96];  
 DEST[63:48]  $\leftarrow$  (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48];  
 TMP[79:64]  $\leftarrow$  (SRC[31:0] < 0) ? 0 : SRC[15:0];  
 DEST[79:64]  $\leftarrow$  (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64];  
 TMP[95:80]  $\leftarrow$  (SRC[63:32] < 0) ? 0 : SRC[47:32];  
 DEST[95:80]  $\leftarrow$  (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80];  
 TMP[111:96]  $\leftarrow$  (SRC[95:64] < 0) ? 0 : SRC[79:64];  
 DEST[111:96]  $\leftarrow$  (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96];  
 TMP[127:112]  $\leftarrow$  (SRC[127:96] < 0) ? 0 : SRC[111:96];  
 DEST[127:112]  $\leftarrow$  (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112];  
 DEST[MAX\_VL-1:128] (Unmodified)

**PACKUSDW (VEX.128 encoded version)**

TMP[15:0]  $\leftarrow$  (SRC1[31:0] < 0) ? 0 : SRC1[15:0];  
 DEST[15:0]  $\leftarrow$  (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];  
 TMP[31:16]  $\leftarrow$  (SRC1[63:32] < 0) ? 0 : SRC1[47:32];  
 DEST[31:16]  $\leftarrow$  (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];  
 TMP[47:32]  $\leftarrow$  (SRC1[95:64] < 0) ? 0 : SRC1[79:64];  
 DEST[47:32]  $\leftarrow$  (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];  
 TMP[63:48]  $\leftarrow$  (SRC1[127:96] < 0) ? 0 : SRC1[111:96];  
 DEST[63:48]  $\leftarrow$  (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];  
 TMP[79:64]  $\leftarrow$  (SRC2[31:0] < 0) ? 0 : SRC2[15:0];  
 DEST[79:64]  $\leftarrow$  (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];  
 TMP[95:80]  $\leftarrow$  (SRC2[63:32] < 0) ? 0 : SRC2[47:32];  
 DEST[95:80]  $\leftarrow$  (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];  
 TMP[111:96]  $\leftarrow$  (SRC2[95:64] < 0) ? 0 : SRC2[79:64];  
 DEST[111:96]  $\leftarrow$  (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];  
 TMP[127:112]  $\leftarrow$  (SRC2[127:96] < 0) ? 0 : SRC2[111:96];  
 DEST[127:112]  $\leftarrow$  (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];  
 DEST[MAX\_VL-1:128]  $\leftarrow$  0;

**VPACKUSDW (VEX.256 encoded version)**

TMP[15:0]  $\leftarrow$  (SRC1[31:0] < 0) ? 0 : SRC1[15:0];  
 DEST[15:0]  $\leftarrow$  (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];  
 TMP[31:16]  $\leftarrow$  (SRC1[63:32] < 0) ? 0 : SRC1[47:32];  
 DEST[31:16]  $\leftarrow$  (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];  
 TMP[47:32]  $\leftarrow$  (SRC1[95:64] < 0) ? 0 : SRC1[79:64];  
 DEST[47:32]  $\leftarrow$  (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];  
 TMP[63:48]  $\leftarrow$  (SRC1[127:96] < 0) ? 0 : SRC1[111:96];  
 DEST[63:48]  $\leftarrow$  (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];  
 TMP[79:64]  $\leftarrow$  (SRC2[31:0] < 0) ? 0 : SRC2[15:0];  
 DEST[79:64]  $\leftarrow$  (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];  
 TMP[95:80]  $\leftarrow$  (SRC2[63:32] < 0) ? 0 : SRC2[47:32];  
 DEST[95:80]  $\leftarrow$  (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];  
 TMP[111:96]  $\leftarrow$  (SRC2[95:64] < 0) ? 0 : SRC2[79:64];  
 DEST[111:96]  $\leftarrow$  (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];

```

TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128];
TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144];
TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160];
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176];
TMP[207:192] ← (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] ← (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192];
TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];
DEST[MAX_VL-1:256] ← 0;

```

**VPACKUSDW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO ((KL/2) - 1)

  i ← j \* 32

  IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

    THEN

      TMP\_SRC2[j+31:i] ← SRC2[31:0]

    ELSE

      TMP\_SRC2[j+31:i] ← SRC2[j+31:i]

  FI;

ENDFOR;

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] ← (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] ← (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];
TMP[111:96] ← (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] ← (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] ← (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
IF VL >= 256
  TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
  DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128];
  TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
  DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144];

```

```

TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160];
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176];
TMP[207:192] ← (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
DEST[207:192] ← (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192];
TMP[223:208] ← (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
DEST[223:208] ← (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
DEST[239:224] ← (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
DEST[255:240] ← (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];

```

FI;

IF VL >= 512

```

TMP[271:256] ← (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
DEST[271:256] ← (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256];
TMP[287:272] ← (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
DEST[287:272] ← (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272];
TMP[303:288] ← (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
DEST[303:288] ← (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288];
TMP[319:304] ← (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
DEST[319:304] ← (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304];
TMP[335:320] ← (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
DEST[335:304] ← (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64];
TMP[351:336] ← (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
DEST[351:336] ← (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336];
TMP[367:352] ← (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
DEST[367:352] ← (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352];
TMP[383:368] ← (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
DEST[383:368] ← (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368];
TMP[399:384] ← (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
DEST[399:384] ← (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384];
TMP[415:400] ← (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
DEST[415:400] ← (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400];
TMP[431:416] ← (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
DEST[431:416] ← (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416];
TMP[447:432] ← (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
DEST[447:432] ← (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432];
TMP[463:448] ← (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
DEST[463:448] ← (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448];
TMP[475:464] ← (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
DEST[475:464] ← (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464];
TMP[491:476] ← (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
DEST[491:476] ← (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476];
TMP[511:492] ← (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
DEST[511:492] ← (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492];

```

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ;merging-masking

```

        THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*           ; zeroing-masking
          DEST[j+15:i] ← 0
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKUSDW__m512i__mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i__mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i__mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i__mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i__mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i__mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i__mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i__mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i__mm256_packus_epi32(__m256i m1, __m256i m2);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

## PACKUSWB—Pack with Unsigned Saturation

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|-----------------------------|--------------------------|--|
| 66 0F 67 /r<br>PACKUSWB <i>xmm1, xmm2/m128</i>                                    | RM         | V/V                         | SSE2                     | Converts 8 signed word integers from <i>xmm1</i> and 8 signed word integers from <i>xmm2/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.                     |
| VEX.NDS.128.66.0F 67 /r<br>VPACKUSWB <i>xmm1,xmm2,<br/>xmm3/m128</i>              | RVM        | V/V                         | AVX                      | Converts 8 signed word integers from <i>xmm2</i> and 8 signed word integers from <i>xmm3/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.                     |
| VEX.NDS.256.66.0F 67 /r<br>VPACKUSWB <i>ymm1, ymm2,<br/>ymm3/m256</i>             | RVM        | V/V                         | AVX2                     | Converts 16 signed word integers from <i>ymm2</i> and 16 signed word integers from <i>ymm3/m256</i> into 32 unsigned byte integers in <i>ymm1</i> using unsigned saturation.                   |
| EVEX.NDS.128.66.0F.WIG 67 /r<br>VPACKUSWB <i>xmm1{k1}{z},<br/>xmm2, xmm3/m128</i> | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Converts signed word integers from <i>xmm2</i> and signed word integers from <i>xmm3/m128</i> into unsigned byte integers in <i>xmm1</i> using unsigned saturation under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG 67 /r<br>VPACKUSWB <i>ymm1{k1}{z},<br/>ymm2, ymm3/m256</i> | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Converts signed word integers from <i>ymm2</i> and signed word integers from <i>ymm3/m256</i> into unsigned byte integers in <i>ymm1</i> using unsigned saturation under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.WIG 67 /r<br>VPACKUSWB <i>zmm1{k1}{z},<br/>zmm2, zmm3/m512</i> | FVM        | V/V                         | AVX512BW                 | Converts signed word integers from <i>zmm2</i> and signed word integers from <i>zmm3/m512</i> into unsigned byte integers in <i>zmm1</i> using unsigned saturation under writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Converts 8, 16 or 32 signed word integers from the first source operand and 8, 16 or 32 signed word integers from the second source operand into 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****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]);
DEST[MAX_VL-1:128] (Unmodified)

```

**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[MAX_VL-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]);  
 DEST[MAX\_VL-1:256] ← 0;

#### VPACKUSWB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

TMP\_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);  
 TMP\_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);  
 TMP\_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);  
 TMP\_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);  
 TMP\_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);  
 TMP\_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);  
 TMP\_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);  
 TMP\_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);  
 TMP\_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);  
 TMP\_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);  
 TMP\_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);  
 TMP\_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);  
 TMP\_DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);  
 TMP\_DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);  
 TMP\_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);  
 TMP\_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);  
 IF VL >= 256  
   TMP\_DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);  
   TMP\_DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);  
   TMP\_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);  
   TMP\_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);  
   TMP\_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);  
   TMP\_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);  
   TMP\_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);  
   TMP\_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);  
   TMP\_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);  
   TMP\_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);  
   TMP\_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);  
   TMP\_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);  
   TMP\_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);  
   TMP\_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);  
   TMP\_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);

```

    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);
    TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
            ELSE *zeroing-masking*                ; zeroing-masking
                DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

VPACKUSWB\_\_m512i\_\_mm512\_packus\_epi16(\_\_m512i m1, \_\_m512i m2);  
 VPACKUSWB\_\_m512i\_\_mm512\_mask\_packus\_epi16(\_\_m512i s, \_\_mmask64 k, \_\_m512i m1, \_\_m512i m2);  
 VPACKUSWB\_\_m512i\_\_mm512\_maskz\_packus\_epi16(\_\_mmask64 k, \_\_m512i m1, \_\_m512i m2);  
 VPACKUSWB\_\_m256i\_\_mm256\_mask\_packus\_epi16(\_\_m256i s, \_\_mmask32 k, \_\_m256i m1, \_\_m256i m2);  
 VPACKUSWB\_\_m256i\_\_mm256\_maskz\_packus\_epi16(\_\_mmask32 k, \_\_m256i m1, \_\_m256i m2);  
 VPACKUSWB\_\_m128i\_\_mm\_mask\_packus\_epi16(\_\_m128i s, \_\_mmask16 k, \_\_m128i m1, \_\_m128i m2);  
 VPACKUSWB\_\_m128i\_\_mm\_maskz\_packus\_epi16(\_\_mmask16 k, \_\_m128i m1, \_\_m128i m2);  
 PACKUSWB\_\_m128i\_\_mm\_packus\_epi16(\_\_m128i m1, \_\_m128i m2);  
 VPACKUSWB\_\_m256i\_\_mm256\_packus\_epi16(\_\_m256i m1, \_\_m256i m2);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PADDB/PADDW/PADD/PADDQ—Add Packed Integers

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|-----------------------------|--------------------------|---|
| 66 0F FC /r<br>PADDB <i>xmm1</i> , <i>xmm2/m128</i>   | RM         | V/V                         | SSE2                     | Add packed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> .  |
| 66 0F FD /r<br>PADDW <i>xmm1</i> , <i>xmm2/m128</i>   | RM         | V/V                         | SSE2                     | Add packed word integers from <i>xmm2/m128</i> and <i>xmm1</i> .  |
| 66 0F FE /r<br>PADD <i>xmm1</i> , <i>xmm2/m128</i>  | RM         | V/V                         | SSE2                     | Add packed doubleword integers from <i>xmm2/m128</i> and <i>xmm1</i> .  |
| 66 0F D4 /r<br>PADDQ <i>xmm1</i> , <i>xmm2/m128</i>   | RM         | V/V                         | SSE2                     | Add packed quadword integers from <i>xmm2/m128</i> and <i>xmm1</i> .  |
| VEX.NDS.128.66.0F.WIG FC /r<br>VPADDB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>                                    | RVM        | V/V                         | AVX                      | Add packed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store in <i>xmm1</i> .   |
| VEX.NDS.128.66.0F.WIG FD /r<br>VPADDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>                                    | RVM        | V/V                         | AVX                      | Add packed word integers from <i>xmm2</i> , <i>xmm3/m128</i> and store in <i>xmm1</i> .   |
| VEX.NDS.128.66.0F.WIG FE /r<br>VPADD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>                                     | RVM        | V/V                         | AVX                      | Add packed doubleword integers from <i>xmm2</i> , <i>xmm3/m128</i> and store in <i>xmm1</i> .                                       |
| VEX.NDS.128.66.0F.WIG D4 /r<br>VPADDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>                                    | RVM        | V/V                         | AVX                      | Add packed quadword integers from <i>xmm2</i> , <i>xmm3/m128</i> and store in <i>xmm1</i> .   |
| VEX.NDS.256.66.0F.WIG FC /r<br>VPADDB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>                                    | RVM        | V/V                         | AVX2                     | Add packed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> .   |
| VEX.NDS.256.66.0F.WIG FD /r<br>VPADDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>                                    | RVM        | V/V                         | AVX2                     | Add packed word integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .   |
| VEX.NDS.256.66.0F.WIG FE /r<br>VPADD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>                                     | RVM        | V/V                         | AVX2                     | Add packed doubleword integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .                                       |
| VEX.NDS.256.66.0F.WIG D4 /r<br>VPADDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>                                    | RVM        | V/V                         | AVX2                     | Add packed quadword integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .   |
| EVEX.NDS.128.66.0F.WIG FC /r<br>VPADDB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> ,<br><i>xmm3/m128</i>       | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Add packed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store in <i>xmm1</i> using writemask <i>k1</i> .               |
| EVEX.NDS.128.66.0F.WIG FD /r<br>VPADDW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> ,<br><i>xmm3/m128</i>       | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Add packed word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store in <i>xmm1</i> using writemask <i>k1</i> .               |
| EVEX.NDS.128.66.0F.W0 FE /r<br>VPADD <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> ,<br><i>xmm3/m128/m32bcst</i> | FV         | V/V                         | AVX512VL<br>AVX512F      | Add packed doubleword integers from <i>xmm2</i> , and <i>xmm3/m128/m32bcst</i> and store in <i>xmm1</i> using writemask <i>k1</i> . |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|-----------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F.W1 D4 /r<br>VPADDQ <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> ,<br><i>xmm3/m128/m64bcst</i> | FV         | V/V                         | AVX512VL<br>AVX512F      | Add packed quadword integers from <i>xmm2</i> , and <i>xmm3/m128/m64bcst</i> and store in <i>xmm1</i> using writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG FC /r<br>VPADDB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> ,<br><i>ymm3/m256</i>        | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Add packed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> using writemask <i>k1</i> .             |
| EVEX.NDS.256.66.0F.WIG FD /r<br>VPADDW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> ,<br><i>ymm3/m256</i>        | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Add packed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> using writemask <i>k1</i> .             |
| EVEX.NDS.256.66.0F.W0 FE /r<br>VPADD <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> ,<br><i>ymm3/m256/m32bcst</i>  | FV         | V/V                         | AVX512VL<br>AVX512F      | Add packed doubleword integers from <i>ymm2</i> , <i>ymm3/m256/m32bcst</i> and store in <i>ymm1</i> using writemask <i>k1</i> .   |
| EVEX.NDS.256.66.0F.W1 D4 /r<br>VPADDQ <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> ,<br><i>ymm3/m256/m64bcst</i> | FV         | V/V                         | AVX512VL<br>AVX512F      | Add packed quadword integers from <i>ymm2</i> , <i>ymm3/m256/m64bcst</i> and store in <i>ymm1</i> using writemask <i>k1</i> .     |
| EVEX.NDS.512.66.0F.WIG FC /r<br>VPADDB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> ,<br><i>zmm3/m512</i>        | FVM        | V/V                         | AVX512BW                 | Add packed byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store in <i>zmm1</i> using writemask <i>k1</i> .             |
| EVEX.NDS.512.66.0F.WIG FD /r<br>VPADDW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> ,<br><i>zmm3/m512</i>        | FVM        | V/V                         | AVX512BW                 | Add packed word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store in <i>zmm1</i> using writemask <i>k1</i> .             |
| EVEX.NDS.512.66.0F.W0 FE /r<br>VPADD <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> ,<br><i>zmm3/m512/m32bcst</i>  | FV         | V/V                         | AVX512F                  | Add packed doubleword integers from <i>zmm2</i> , <i>zmm3/m512/m32bcst</i> and store in <i>zmm1</i> using writemask <i>k1</i> .   |
| EVEX.NDS.512.66.0F.W1 D4 /r<br>VPADDQ <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> ,<br><i>zmm3/m512/m64bcst</i> | FV         | V/V                         | AVX512F                  | Add packed quadword integers from <i>zmm2</i> , <i>zmm3/m512/m64bcst</i> and store in <i>zmm1</i> using writemask <i>k1</i> .     |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDQ: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### PADDB (Legacy SSE instruction)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] ← DEST[127:120] + SRC[127:120];
DEST[MAX_VL-1:128] (Unmodified)
```

### PADDW (Legacy SSE instruction)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] ← DEST[127:112] + SRC[127:112];
DEST[MAX_VL-1:128] (Unmodified)
```

### PADDD (Legacy SSE instruction)

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← DEST[127:96] + SRC[127:96];
DEST[MAX_VL-1:128] (Unmodified)
```

### PADDQ (Legacy SSE instruction)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

DEST[127:64] ← DEST[127:64] + SRC[127:64];  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPADDB (VEX.128 encoded instruction)**

DEST[7:0] ← SRC1[7:0] + SRC2[7:0];  
 (\* Repeat add operation for 2nd through 15th byte \*)  
 DEST[127:120] ← SRC1[127:120] + SRC2[127:120];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDW (VEX.128 encoded instruction)**

DEST[15:0] ← SRC1[15:0] + SRC2[15:0];  
 (\* Repeat add operation for 2nd through 7th word \*)  
 DEST[127:112] ← SRC1[127:112] + SRC2[127:112];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADD (VEX.128 encoded instruction)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0];  
 (\* Repeat add operation for 2nd and 3th doubleword \*)  
 DEST[127:96] ← SRC1[127:96] + SRC2[127:96];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDQ (VEX.128 encoded instruction)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0];  
 DEST[127:64] ← SRC1[127:64] + SRC2[127:64];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDB (VEX.256 encoded instruction)**

DEST[7:0] ← SRC1[7:0] + SRC2[7:0];  
 (\* Repeat add operation for 2nd through 31th byte \*)  
 DEST[255:248] ← SRC1[255:248] + SRC2[255:248];

**VPADDW (VEX.256 encoded instruction)**

DEST[15:0] ← SRC1[15:0] + SRC2[15:0];  
 (\* Repeat add operation for 2nd through 15th word \*)  
 DEST[255:240] ← SRC1[255:240] + SRC2[255:240];

**VPADD (VEX.256 encoded instruction)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0];  
 (\* Repeat add operation for 2nd and 7th doubleword \*)  
 DEST[255:224] ← SRC1[255:224] + SRC2[255:224];

**VPADDQ (VEX.256 encoded instruction)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0];  
 DEST[127:64] ← SRC1[127:64] + SRC2[127:64];  
 DEST[191:128] ← SRC1[191:128] + SRC2[191:128];  
 DEST[255:192] ← SRC1[255:192] + SRC2[255:192];

**VPADDB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]

```

ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+7:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
        DEST[j+7:i] = 0
    FI
FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPADDW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[i+15:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPADD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
            ELSE DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[j+31:i] remains unchanged*
            ELSE *zeroing-masking*         ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```



**VPADDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPADDB \_\_m512i\_mm512\_add\_epi8 (\_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i\_mm512\_add\_epi16 (\_\_m512i a, \_\_m512i b)

VPADDB \_\_m512i\_mm512\_mask\_add\_epi8 (\_\_m512i s, \_\_mmask64 m, \_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i\_mm512\_mask\_add\_epi16 (\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i b)

VPADDB \_\_m512i\_mm512\_maskz\_add\_epi8 (\_\_mmask64 m, \_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i\_mm512\_maskz\_add\_epi16 (\_\_mmask32 m, \_\_m512i a, \_\_m512i b)

VPADDB \_\_m256i\_mm256\_mask\_add\_epi8 (\_\_m256i s, \_\_mmask32 m, \_\_m256i a, \_\_m256i b)

VPADDW \_\_m256i\_mm256\_mask\_add\_epi16 (\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i b)

VPADDB \_\_m256i\_mm256\_maskz\_add\_epi8 (\_\_mmask32 m, \_\_m256i a, \_\_m256i b)

VPADDW \_\_m256i\_mm256\_maskz\_add\_epi16 (\_\_mmask16 m, \_\_m256i a, \_\_m256i b)

VPADDB \_\_m128i\_mm\_mask\_add\_epi8 (\_\_m128i s, \_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPADDW \_\_m128i\_mm\_mask\_add\_epi16 (\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPADDB \_\_m128i\_mm\_maskz\_add\_epi8 (\_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPADDW \_\_m128i\_mm\_maskz\_add\_epi16 (\_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPADDD \_\_m512i\_mm512\_add\_epi32 (\_\_m512i a, \_\_m512i b);

VPADDD \_\_m512i\_mm512\_mask\_add\_epi32 (\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPADDD \_\_m512i\_mm512\_maskz\_add\_epi32 (\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPADDD \_\_m256i\_mm256\_mask\_add\_epi32 (\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDD \_\_m256i\_mm256\_maskz\_add\_epi32 (\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDD \_\_m128i\_mm\_mask\_add\_epi32 (\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDD \_\_m128i\_mm\_maskz\_add\_epi32 (\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDQ \_\_m512i\_mm512\_add\_epi64 (\_\_m512i a, \_\_m512i b);

VPADDQ \_\_m512i\_mm512\_mask\_add\_epi64 (\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPADDQ \_\_m512i\_mm512\_maskz\_add\_epi64 (\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPADDQ \_\_m256i\_mm256\_mask\_add\_epi64 (\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDQ \_\_m256i\_mm256\_maskz\_add\_epi64 (\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDQ \_\_m128i\_mm\_mask\_add\_epi64 (\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDQ \_\_m128i\_mm\_maskz\_add\_epi64 (\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PADDB \_\_m128i\_mm\_add\_epi8 (\_\_m128i a, \_\_m128i b)

PADDW \_\_m128i\_mm\_add\_epi16 (\_\_m128i a, \_\_m128i b)

PADDD \_\_m128i\_mm\_add\_epi32 (\_\_m128i a, \_\_m128i b)

PADDDQ \_\_m128i\_mm\_add\_epi64 (\_\_m128i a, \_\_m128i b)

VPADDB \_\_m256i\_mm256\_add\_epi8 (\_\_m256ia, \_\_m256i b)

VPADDW \_\_m256i \_\_mm256\_add\_epi16 ( \_\_m256i a, \_\_m256i b)  
VPADDQ \_\_m256i \_\_mm256\_add\_epi32 ( \_\_m256i a, \_\_m256i b)  
VPADDQ \_\_m256i \_\_mm256\_add\_epi64 ( \_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.  
EVEX-encoded VPADDQ, see Exceptions Type E4.  
EVEX-encoded VPADDQ, see Exceptions Type E4.nb.

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F EC /r<br>PADDSB <i>xmm1</i> , <i>xmm2/m128</i>   | RM         | V/V                          | SSE2                     | Add packed signed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.  |
| 66 0F ED /r<br>PADDSW <i>xmm1</i> , <i>xmm2/m128</i>   | RM         | V/V                          | SSE2                     | Add packed signed word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.  |
| VEX.NDS.128.66.0F EC<br>VPADDSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>                                     | RVM        | V/V                          | AVX                      | Add packed signed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> .                           |
| VEX.NDS.128.66.0F ED<br>VPADDSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>                                     | RVM        | V/V                          | AVX                      | Add packed signed word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> .                           |
| VEX.NDS.256.66.0F EC<br>VPADDSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>                                     | RVM        | V/V                          | AVX2                     | Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .                           |
| VEX.NDS.256.66.0F ED<br>VPADDSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>                                     | RVM        | V/V                          | AVX2                     | Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .                           |
| EVEX.NDS.128.66.0F.WIG EC /r<br>VPADDSB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> ,<br><i>xmm3/m128</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed signed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG EC /r<br>VPADDSB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> ,<br><i>ymm3/m256</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.WIG EC /r<br>VPADDSB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> ,<br><i>zmm3/m512</i> | FVM        | V/V                          | AVX512BW                 | Add packed signed byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.128.66.0F.WIG ED /r<br>VPADDSW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> ,<br><i>xmm3/m128</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed signed word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG ED /r<br>VPADDSW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> ,<br><i>ymm3/m256</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.WIG ED /r<br>VPADDSW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> ,<br><i>zmm3/m512</i> | FVM        | V/V                          | AVX512BW                 | Add packed signed word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

(V)PADDQB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDQW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****PADDQB (Legacy SSE instruction)**

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 15th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] + SRC[127:120]);
DEST[MAX_VL-1:128] (Unmodified)
```

**PADDQW (Legacy SSE instruction)**

```
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]);
DEST[MAX_VL-1:128] (Unmodified)
```

**VPADDQB (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 15th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] + SRC2[127:120]);
DEST[MAX_VL-1:128] ← 0
```

**VPADDQW (VEX.128 encoded version)**

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[MAX_VL-1:128] ← 0
```

**VPADDQB (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]);
DEST[MAX_VL-1:256] ← 0
```

**VPADDQW (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])
DEST[MAX_VL-1:256] ← 0
```

**VPADDSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**VPADDSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalents**

```
VPADDSB__m512i__mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW__m512i__mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB__m512i__mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i__mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB__m512i__mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i__mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB__m256i__mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i__mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB__m256i__mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i__mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB__m128i__mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i__mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB__m128i__mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i__mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)
PADDSB__m128i__mm_adds_epi8 (__m128i a, __m128i b)
```

PADDSW\_\_m128i\_mm\_adds\_epi16 (\_\_m128i a, \_\_m128i b)  
VPADDSB\_\_m128i\_mm\_adds\_epi8 (\_\_m128i a, \_\_m128i b)  
VPADDSW\_\_m128i\_mm\_adds\_epi16 (\_\_m128i a, \_\_m128i b)  
VPADDSB\_\_m256i\_mm256\_adds\_epi8 (\_\_m256i a, \_\_m256i b)  
VPADDSW\_\_m256i\_mm256\_adds\_epi16 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F DC /r<br>PADDUSB <i>xmm1, xmm2/m128</i>                                 | RM         | V/V                          | SSE2                     | Add packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.  |
| 66 0F DD /r<br>PADDUSW <i>xmm1, xmm2/m128</i>                                 | RM         | V/V                          | SSE2                     | Add packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.  |
| VEX.NDS.128.66.0F DC<br>VPADDUSB <i>xmm1, xmm2, xmm3/m128</i>                 | RVM        | V/V                          | AVX                      | Add packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> .                           |
| VEX.NDS.128.66.0F DD<br>VPADDUSW <i>xmm1, xmm2, xmm3/m128</i>                 | RVM        | V/V                          | AVX                      | Add packed unsigned word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> .                           |
| VEX.NDS.256.66.0F DC<br>VPADDUSB <i>ymm1, ymm2, ymm3/m256</i>                 | RVM        | V/V                          | AVX2                     | Add packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .                           |
| VEX.NDS.256.66.0F DD<br>VPADDUSW <i>ymm1, ymm2, ymm3/m256</i>                 | RVM        | V/V                          | AVX2                     | Add packed unsigned word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .                           |
| EVEX.NDS.128.66.0F.WIG DC /r<br>VPADDUSB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG DC /r<br>VPADDUSB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.WIG DC /r<br>VPADDUSB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i> | FVM        | V/V                          | AVX512BW                 | Add packed unsigned byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.128.66.0F.WIG DD /r<br>VPADDUSW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed unsigned word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG DD /r<br>VPADDUSW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Add packed unsigned word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.WIG DD /r<br>VPADDUSW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i> | FVM        | V/V                          | AVX512BW                 | Add packed unsigned word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****PADDUSB (Legacy SSE instruction)**

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 15th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] + SRC[127:120]);
DEST[MAX_VL-1:128] (Unmodified)
```

**PADDUSW (Legacy SSE instruction)**

```
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]);
DEST[MAX_VL-1:128] (Unmodified)
```

**VPADDUSB (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 15th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] + SRC2[127:120]);
DEST[MAX_VL-1:128] ← 0
```

**VPADDUSW (VEX.128 encoded version)**

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[MAX_VL-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]);
DEST[MAX_VL-1:256] ← 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])
DEST[MAX_VL-1:256] ← 0
```

VPADDUSB (EVEX encoded versions)  
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**VPADDUSW (EVEX encoded versions)**  
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
VPADDUSB__m512i__mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW__m512i__mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB__m512i__mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i__mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB__m512i__mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i__mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB__m256i__mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i__mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB__m256i__mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i__mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB__m128i__mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i__mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB__m128i__mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i__mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)
PADDUSB__m128i__mm_adds_epu8 (__m128i a, __m128i b)
```

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)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PALIGNR—Byte Align

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F 3A 0F /r ib<br>PALIGNR xmm1, xmm2/m128, imm8                                   | RM         | V/V                          | SSSE3                    | Concatenate destination and source operands, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.  |
| VEX.NDS.128.66.0F3A 0F /r ib<br>VPALIGNR xmm1, xmm2,<br>xmm3/m128, imm8              | RVM        | V/V                          | AVX                      | Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.  |
| VEX.NDS.256.66.0F3A 0F /r ib<br>VPALIGNR ymm1, ymm2,<br>ymm3/m256, imm8              | RVM        | V/V                          | AVX2                     | Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.  |
| EVEX.NDS.128.66.0F3A.WIG 0F /r ib<br>VPALIGNR xmm1 {k1}{z}, xmm2,<br>xmm3/m128, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.  |
| EVEX.NDS.256.66.0F3A.WIG 0F /r ib<br>VPALIGNR ymm1 {k1}{z}, ymm2,<br>ymm3/m256, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.  |
| EVEX.NDS.512.66.0F3A.WIG 0F /r ib<br>VPALIGNR zmm1 {k1}{z}, zmm2,<br>zmm3/m512, imm8 | FVM        | V/V                          | AVX512BW                 | Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

#### Description

PALIGNR concatenates the first source operand and the second source 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 immediate value is considered unsigned. Immediate shift counts larger than 32 for 128-bit operands produces a zero result.

Legacy SSE instructions: In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bit of the intermediate composite 256-bit result came from the 128-bit data from the first source operand, the low 128-bit of the intermediate result came from the 128-bit data of the second source operand. In the same way, the 512-bit encoded version produces results on a block of 16-byte basis.

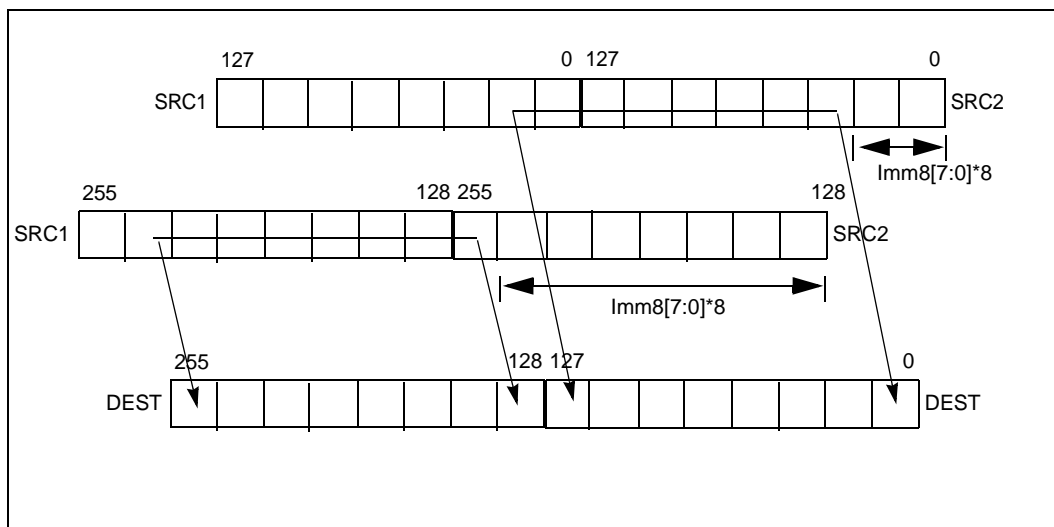


Figure 5-29. 256-bit VPALIGN Instruction Operation

**Operation**

**PALIGNR**

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAX_VL-1:128] (Unmodified)
```

**VPALIGNR (VEX.128 encoded versions)**

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAX_VL-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]
DEST[MAX_VL-1:256] ← 0
```

**VPALIGNR (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR I ← 0 TO VL-1 with increments of 128
    temp1[255:0] ← ((SRC1[I+127:I] << 128) OR SRC2[I+127:I]) >> (imm8[7:0]*8);
    TMP_DEST[I+127:I] ← temp1[127:0]
ENDFOR;

```

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPALIGNR __m512i __mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i __mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i __mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i __mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i __mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i __mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i __mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)
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**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

**PAND—Logical AND**

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| 66 0F DB /r<br>PAND xmm1, xmm2/m128  | RM        | V/V                          | SSE2                     | Bitwise AND of xmm2/m128 and xmm1.   |
| VEX.NDS.128.66.0F.WIG DB /r<br>VPAND xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX                      | Bitwise AND of xmm2, and xmm3/m128 and store result in xmm1.   |
| VEX.NDS.256.66.0F.WIG DB /r<br>VPAND ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.   |
| EVEX.NDS.128.66.0F.W0 DB /r<br>VPANDD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F.W0 DB /r<br>VPANDD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F.W0 DB /r<br>VPANDD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1. |
| EVEX.NDS.128.66.0F.W1 DB /r<br>VPANDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.   |
| EVEX.NDS.256.66.0F.W1 DB /r<br>VPANDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.   |
| EVEX.NDS.512.66.0F.W1 DB /r<br>VPANDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.   |

**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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### PAND (Legacy SSE instruction)

DEST[127:0] ← (DEST[127:0] BITWISE AND SRC[127:0])

### VPAND (VEX.128 encoded instruction)

DEST[127:0] ← (SRC1[127:0] AND SRC2[127:0])

DEST[MAX\_VL-1:128] ← 0

### VPAND (VEX.256 encoded instruction)

DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])

DEST[MAX\_VL-1:256] ← 0

### VPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]

        ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPANDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPANDD \_\_m512i \_\_mm512\_and\_epi32(\_\_m512i a, \_\_m512i b);

VPANDD \_\_m512i \_\_mm512\_mask\_and\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDD \_\_m512i \_\_mm512\_maskz\_and\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDQ \_\_m512i \_\_mm512\_and\_epi64(\_\_m512i a, \_\_m512i b);

VPANDQ \_\_m512i \_\_mm512\_mask\_and\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDQ \_\_m512i \_\_mm512\_maskz\_and\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m256i \_\_mm256\_mask\_and\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m256i \_\_mm256\_maskz\_and\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m128i \_\_mm\_mask\_and\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDND \_\_m128i \_\_mm\_maskz\_and\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m256i \_\_mm256\_mask\_and\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m256i \_\_mm256\_maskz\_and\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m128i \_\_mm\_mask\_and\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m128i \_\_mm\_maskz\_and\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PAND \_\_m128i \_\_mm\_and\_si128 (\_\_m128i a, \_\_m128i b)

VPAND \_\_m256i \_\_mm256\_and\_si256 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## PANDN—Logical AND NOT

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| 66 0F DF /r<br>PANDN xmm1, xmm2/m128  | RM        | V/V                          | SSE2                     | Bitwise AND NOT of xmm2/m128 and xmm1.   |
| VEX.NDS.128.66.0F.WIG DF /r<br>VPANDN xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX                      | Bitwise AND NOT of xmm2, and xmm3/m128 and store result in xmm1.   |
| VEX.NDS.256.66.0F.WIG DF /r<br>VPANDN ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.   |
| EVEX.NDS.128.66.0F.W0 DF /r<br>VPANDND xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F.W0 DF /r<br>VPANDND ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F.W0 DF /r<br>VPANDND zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1. |
| EVEX.NDS.128.66.0F.W1 DF /r<br>VPANDNQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.   |
| EVEX.NDS.256.66.0F.W1 DF /r<br>VPANDNQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.   |
| EVEX.NDS.512.66.0F.W1 DF /r<br>VPANDNQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.   |

### 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1; otherwise, it is set to 0.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### PANDN (Legacy SSE instruction)

DEST[127:0] ← ((NOT DEST[127:0]) AND SRC[127:0])

### VPANDN (VEX.128 encoded instruction)

DEST[127:0] ← (( NOT SRC1[127:0]) AND SRC2[127:0])

DEST[MAX\_VL-1:128] ← 0

### VPANDN (VEX.256 encoded instruction)

DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])

DEST[MAX\_VL-1:256] ← 0

### VPANDND (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[31:0])

        ELSE DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[i+31:i])

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPANDNQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])

ELSE DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPANDND \_\_m512i \_\_mm512\_andnot\_epi32( \_\_m512i a, \_\_m512i b);

VPANDND \_\_m512i \_\_mm512\_mask\_andnot\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m512i \_\_mm512\_maskz\_andnot\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m256i \_\_mm256\_mask\_andnot\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m256i \_\_mm256\_maskz\_andnot\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m128i \_\_mm\_mask\_andnot\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDND \_\_m128i \_\_mm\_maskz\_andnot\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m512i \_\_mm512\_andnot\_epi64( \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m512i \_\_mm512\_mask\_andnot\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m512i \_\_mm512\_maskz\_andnot\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m256i \_\_mm256\_mask\_andnot\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m256i \_\_mm256\_maskz\_andnot\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m128i \_\_mm\_mask\_andnot\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m128i \_\_mm\_maskz\_andnot\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PANDN \_\_m128i \_\_mm\_andnot\_si128( \_\_m128i a, \_\_m128i b)

VPANDN \_\_m256i \_\_mm256\_andnot\_si256( \_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PAVGB/PAVGW—Average Packed Integers

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F E0, /r<br>PAVGB <i>xmm1, xmm2/m128</i>                                | RM         | V/V                          | SSE2                     | Average packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.   |
| 66 0F E3, /r<br>PAVGW <i>xmm1, xmm2/m128</i>                                | RM         | V/V                          | SSE2                     | Average packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.   |
| VEX.NDS.128.66.0F E0<br>VPAVGB <i>xmm1, xmm2, xmm3/m128</i>                 | RVM        | V/V                          | AVX                      | Average packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> with rounding and store to <i>xmm1</i> .                           |
| VEX.NDS.128.66.0F E3<br>VPAVGW <i>xmm1, xmm2, xmm3/m128</i>                 | RVM        | V/V                          | AVX                      | Average packed unsigned word integers from <i>xmm2</i> , <i>xmm3/m128</i> with rounding to <i>xmm1</i> .   |
| VEX.NDS.256.66.0F E0<br>VPAVGB <i>ymm1, ymm2, ymm3/m256</i>                 | RVM        | V/V                          | AVX2                     | Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> .                           |
| VEX.NDS.256.66.0F E3<br>VPAVGW <i>ymm1, ymm2, ymm3/m256</i>                 | RVM        | V/V                          | AVX2                     | Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> .   |
| EVEX.NDS.128.66.0F.WIG E0 /r<br>VPAVGB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Average packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> with rounding and store to <i>xmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.256.66.0F.WIG E0 /r<br>VPAVGB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.512.66.0F.WIG E0 /r<br>VPAVGB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i> | FVM        | V/V                          | AVX512BW                 | Average packed unsigned byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> with rounding and store to <i>zmm1</i> under writemask <i>k1</i> . |
| EVEX.NDS.128.66.0F.WIG E3 /r<br>VPAVGW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Average packed unsigned word integers from <i>xmm2</i> , <i>xmm3/m128</i> with rounding to <i>xmm1</i> under writemask <i>k1</i> .               |
| EVEX.NDS.256.66.0F.WIG E3 /r<br>VPAVGW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i> | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> under writemask <i>k1</i> .               |
| EVEX.NDS.512.66.0F.WIG E3 /r<br>VPAVGW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i> | FVM        | V/V                          | AVX512BW                 | Average packed unsigned word integers from <i>zmm2</i> , <i>zmm3/m512</i> with rounding to <i>zmm1</i> under writemask <i>k1</i> .               |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Performs a SIMD average of the packed unsigned integers from the second source operand and the 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.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

## Operation

### PAVGB (Legacy SSE instruction)

```
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 15 )
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1;
DEST[MAX_VL-1:128] unmodified;
```

### PAVGW (Legacy SSE instruction)

```
SRC[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 7 )
DEST[127:48] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
DEST[MAX_VL-1:128] unmodified;
```

### VPAVGB (VEX.128 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 15 )
DEST[127:48] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1;
DEST[MAX_VL-1:128] ← 0
```

### VPAVGW (VEX.128 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 7 )
DEST[127:4] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1;
DEST[MAX_VL-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;
DEST[MAX_VL-1:256] ← 0
```

### 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;
```

DEST[MAX\_VL-1:256] ← 0

VPAVGB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (\* Temp sum before shifting is 9 bits \*)

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPAVGW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1  
; (\* Temp sum before shifting is 17 bits \*)

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB \_\_m512i\_mm512\_avg\_epu8(\_\_m512i a, \_\_m512i b);

VPAVGW \_\_m512i\_mm512\_avg\_epu16(\_\_m512i a, \_\_m512i b);

VPAVGB \_\_m512i\_mm512\_mask\_avg\_epu8(\_\_m512i s, \_\_mmask64 m, \_\_m512i a, \_\_m512i b);

VPAVGW \_\_m512i\_mm512\_mask\_avg\_epu16(\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i b);

VPAVGB \_\_m512i\_mm512\_maskz\_avg\_epu8(\_\_mmask64 m, \_\_m512i a, \_\_m512i b);

VPAVGW \_\_m512i\_mm512\_maskz\_avg\_epu16(\_\_mmask32 m, \_\_m512i a, \_\_m512i b);

VPAVGB \_\_m256i\_mm256\_mask\_avg\_epu8(\_\_m256i s, \_\_mmask32 m, \_\_m256i a, \_\_m256i b);

VPAVGW \_\_m256i\_mm256\_mask\_avg\_epu16(\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i b);

VPAVGB \_\_m256i\_mm256\_maskz\_avg\_epu8(\_\_mmask32 m, \_\_m256i a, \_\_m256i b);

VPAVGW \_\_m256i\_mm256\_maskz\_avg\_epu16(\_\_mmask16 m, \_\_m256i a, \_\_m256i b);

VPAVGB \_\_m128i\_mm\_mask\_avg\_epu8(\_\_m128i s, \_\_mmask16 m, \_\_m128i a, \_\_m128i b);

VPAVGW \_\_m128i\_mm\_mask\_avg\_epu16(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b);

VPAVGB \_\_m128i\_mm\_maskz\_avg\_epu8(\_\_mmask16 m, \_\_m128i a, \_\_m128i b);

VPAVGW \_\_m128i\_mm\_maskz\_avg\_epu16(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

PAVGB \_\_m128i\_mm\_avg\_epu8(\_\_m128i a, \_\_m128i b)

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)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPBROADCASTM—Broadcast Mask to Vector Register

| Opcode/<br>Instruction                                | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F38.W1 2A /r<br>VPBROADCASTMB2Q xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512CD     | Broadcast low byte value in k1 to two locations in xmm1.     |
| EVEX.256.F3.0F38.W1 2A /r<br>VPBROADCASTMB2Q ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512CD     | Broadcast low byte value in k1 to four locations in ymm1.    |
| EVEX.512.F3.0F38.W1 2A /r<br>VPBROADCASTMB2Q zmm1, k1 | RM        | V/V                          | AVX512CD                 | Broadcast low byte value in k1 to eight locations in zmm1.   |
| EVEX.128.F3.0F38.W0 3A /r<br>VPBROADCASTMW2D xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512CD     | Broadcast low word value in k1 to four locations in xmm1.    |
| EVEX.256.F3.0F38.W0 3A /r<br>VPBROADCASTMW2D ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512CD     | Broadcast low word value in k1 to eight locations in ymm1.   |
| EVEX.512.F3.0F38.W0 3A /r<br>VPBROADCASTMW2D zmm1, k1 | RM        | V/V                          | AVX512CD                 | Broadcast low word value in k1 to sixteen locations in zmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VPBROADCASTMB2Q

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j\*64

    DEST[i+63:i] ← ZeroExtend(SRC[7:0])

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VPBROADCASTMW2D

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j\*32

    DEST[i+31:i] ← ZeroExtend(SRC[15:0])

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q \_\_m512i \_\_mm512\_broadcastmb\_epi64(\_\_mmask8);

VPBROADCASTMW2D \_\_m512i \_\_mm512\_broadcastmw\_epi32(\_\_mmask16);



VPBROADCASTMB2Q \_\_m256i \_\_mm256\_broadcastmb\_epi64( \_\_mmask8);  
VPBROADCASTMW2D \_\_m256i \_\_mm256\_broadcastmw\_epi32( \_\_mmask8);  
VPBROADCASTMB2Q \_\_m128i \_\_mm\_broadcastmb\_epi64( \_\_mmask8);  
VPBROADCASTMW2D \_\_m128i \_\_mm\_broadcastmw\_epi32( \_\_mmask8);

#### **SIMD Floating-Point Exceptions**

None

#### **Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6NF.

## PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ—Compare Packed Integers for Equality

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 74 /r<br>PCMPEQB xmm1, xmm2/m128                                      | RM         | V/V                          | SSE2                     | Compare packed bytes in xmm2/m128 and xmm1 for equality.  |
| 66 0F 75 /r<br>PCMPEQW xmm1, xmm2/m128                                      | RM         | V/V                          | SSE2                     | Compare packed words in xmm2/m128 and xmm1 for equality.  |
| 66 0F 76 /r<br>PCMPEQD xmm1, xmm2/m128                                      | RM         | V/V                          | SSE2                     | Compare packed doublewords in xmm2/m128 and xmm1 for equality.  |
| 66 0F 38 29 /r<br>PCMPEQQ xmm1, xmm2/m128                                   | RM         | V/V                          | SSE4_1                   | Compare packed quadwords in xmm2/m128 and xmm1 for equality.  |
| VEX.NDS.128.66.0F.WIG 74 /r<br>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<br>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<br>VPCMPEQD xmm1, xmm2, xmm3/m128               | RVM        | V/V                          | AVX                      | Compare packed doublewords in xmm3/m128 and xmm2 for equality.  |
| VEX.NDS.128.66.0F38.WIG 29 /r<br>VPCMPEQQ xmm1, xmm2, xmm3/m128             | RVM        | V/V                          | AVX                      | Compare packed quadwords in xmm3/m128 and xmm2 for equality.  |
| VEX.NDS.256.66.0F.WIG 74 /r<br>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<br>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<br>VPCMPEQD ymm1, ymm2, ymm3 /m256              | RVM        | V/V                          | AVX2                     | Compare packed doublewords in ymm3/m256 and ymm2 for equality.  |
| VEX.NDS.256.66.0F38.WIG 29 /r<br>VPCMPEQQ ymm1, ymm2, ymm3 /m256            | RVM        | V/V                          | AVX2                     | Compare packed quadwords in ymm3/m256 and ymm2 for equality.  |
| EVEX.NDS.128.66.0F.W0 76 /r<br>VPCMPEQD k1 {k2}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.256.66.0F.W0 76 /r<br>VPCMPEQD k1 {k2}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.512.66.0F.W0 76 /r<br>VPCMPEQD k1 {k2}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2,                                     |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F38.W1 29 /r<br>VPCMPEQQ k1 {k2}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare Equal between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.256.66.0F38.W1 29 /r<br>VPCMPEQQ k1 {k2}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare Equal between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.512.66.0F38.W1 29 /r<br>VPCMPEQQ k1 {k2}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Compare Equal between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.128.66.0F.WIG 74 /r<br>VPCMPEQB k1 {k2}, xmm2, xmm3 /m128            | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.                     |
| EVEX.NDS.256.66.0F.WIG 74 /r<br>VPCMPEQB k1 {k2}, ymm2, ymm3 /m256            | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.                     |
| EVEX.NDS.512.66.0F.WIG 74 /r<br>VPCMPEQB k1 {k2}, zmm2, zmm3 /m512            | FVM        | V/V                          | AVX512BW                 | Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.                     |
| EVEX.NDS.128.66.0F.WIG 75 /r<br>VPCMPEQW k1 {k2}, xmm2, xmm3 /m128            | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.                     |
| EVEX.NDS.256.66.0F.WIG 75 /r<br>VPCMPEQW k1 {k2}, ymm2, ymm3 /m256            | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.                     |
| EVEX.NDS.512.66.0F.WIG 75 /r<br>VPCMPEQW k1 {k2}, zmm2, zmm3 /m512            | FVM        | V/V                          | AVX512BW                 | Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.                     |

### 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)    | VEEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD compare for equality of the packed bytes, words, doublewords, or quadwords in the first source operand and the second source 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 PCMPQB instruction compares the corresponding bytes in the destination and source operands; the PCMPQW instruction compares the corresponding words in the destination and source operands; the PCMPQD instruction compares the corresponding doublewords in the destination and source operands, and the PCMPQQ instruction compares the corresponding quadwords in the destination and source operands.

Legacy SSE instructions: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. 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 operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAX\_VL-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 (MAX\_VL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

## Operation

### COMPARE\_BYTES\_EQUAL (SRC1, SRC2)

```
IF SRC1[7:0] = SRC2[7:0]
THEN DEST[7:0] ← FFH;
ELSE DEST[7:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 \*)

```
IF SRC1[127:120] = SRC2[127:120]
THEN DEST[127:120] ← FFH;
ELSE DEST[127:120] ← 0; FI;
```

### COMPARE\_WORDS\_EQUAL (SRC1, SRC2)

```
IF SRC1[15:0] = SRC2[15:0]
THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 \*)

```
IF SRC1[127:112] = SRC2[127:112]
THEN DEST[127:112] ← FFFFH;
ELSE DEST[127:112] ← 0; FI;
```

### COMPARE\_DWORDS\_EQUAL (SRC1, SRC2)

```
IF SRC1[31:0] = SRC2[31:0]
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 \*)

```
IF SRC1[127:96] = SRC2[127:96]
THEN DEST[127:96] ← FFFFFFFFH;
ELSE DEST[127:96] ← 0; FI;
```

**COMPARE\_QWORDS\_EQUAL (SRC1, SRC2)**

```

IF SRC1[63:0] = SRC2[63:0]
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] = SRC2[127:64]
THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] ← 0; FI;

```

**VPCMPEQB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+7:i] == SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VPCMPEQB (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAX_VL-1:256] ← 0

```

**VPCMPEQB (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ← 0

```

**PCMPEQB (128-bit Legacy SSE version)**

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

```

**VPCMPEQW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+15:i] == SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VPCMPEQW (VEX.256 encoded version)**

```
DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAX_VL-1:256] ← 0
```

**VPCMPEQW (VEX.128 encoded version)**

```
DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ← 0
```

**PCMPEQW (128-bit Legacy SSE version)**

```
DEST[127:0] ← COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

**VPCMPEQD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k2[j] OR \*no writemask\*

    THEN

      /\* signed comparison \*/

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+31:i] = SRC2[31:0];

        ELSE CMP ← SRC1[i+31:i] = SRC2[i+31:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPEQD (VEX.256 encoded version)**

```
DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAX_VL-1:256] ← 0
```

**VPCMPEQD (VEX.128 encoded version)**

```
DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAX_VL-1:128] ← 0
```

**PCMPEQD (128-bit Legacy SSE version)**

```
DEST[127:0] ← COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

**VPCMPEQQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+63:i] = SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] = SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPEQQ (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_QWORDS\_EQUAL(SRC1[127:0],SRC2[127:0])

DEST[255:128] ← COMPARE\_QWORDS\_EQUAL(SRC1[255:128],SRC2[255:128])

DEST[MAX\_VL-1:256] ← 0

**VPCMPEQQ (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_QWORDS\_EQUAL(SRC1[127:0],SRC2[127:0])

DEST[MAX\_VL-1:128] ← 0

**PCMPEQQ (128-bit Legacy SSE version)**

DEST[127:0] ← COMPARE\_QWORDS\_EQUAL(DEST[127:0],SRC[127:0])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPEQB \_\_mmask64 \_\_mm512\_cmpeq\_epi8\_mask(\_\_m512i a, \_\_m512i b);

VPCMPEQB \_\_mmask64 \_\_mm512\_mask\_cmpeq\_epi8\_mask(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPCMPEQB \_\_mmask32 \_\_mm256\_cmpeq\_epi8\_mask(\_\_m256i a, \_\_m256i b);

VPCMPEQB \_\_mmask32 \_\_mm256\_mask\_cmpeq\_epi8\_mask(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPCMPEQB \_\_mmask16 \_\_mm\_cmpeq\_epi8\_mask(\_\_m128i a, \_\_m128i b);

VPCMPEQB \_\_mmask16 \_\_mm\_mask\_cmpeq\_epi8\_mask(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPCMPEQD \_\_mmask16 \_\_mm512\_cmpeq\_epi32\_mask(\_\_m512i a, \_\_m512i b);

VPCMPEQD \_\_mmask16 \_\_mm512\_mask\_cmpeq\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPCMPEQQ \_\_mmask8 \_\_mm512\_cmpeq\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPCMPEQQ \_\_mmask8 \_\_mm512\_mask\_cmpeq\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPCMPEQD \_\_mmask8 \_\_mm256\_cmpeq\_epi32\_mask(\_\_m256i a, \_\_m256i b);

VPCMPEQD \_\_mmask8 \_\_mm256\_mask\_cmpeq\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPCMPEQQ \_\_mmask8 \_\_mm256\_cmpeq\_epi64\_mask(\_\_m256i a, \_\_m256i b);

VPCMPEQQ \_\_mmask8 \_\_mm256\_mask\_cmpeq\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPCMPEQD \_\_mmask8 \_\_mm\_cmpeq\_epi32\_mask(\_\_m128i a, \_\_m128i b);

VPCMPEQD \_\_mmask8 \_\_mm\_mask\_cmpeq\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPCMPEQQ \_\_mmask8 \_\_mm\_cmpeq\_epi64\_mask(\_\_m128i a, \_\_m128i b);

VPCMPEQQ \_\_mmask8 \_\_mm\_mask\_cmpeq\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPCMPEQW \_\_mmask32 \_\_mm512\_cmpeq\_epi16\_mask(\_\_m512i a, \_\_m512i b);

VPCMPEQW \_\_mmask32 \_\_mm512\_mask\_cmpeq\_epi16\_mask(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPCMPEQW \_\_mmask16 \_\_mm256\_cmpeq\_epi16\_mask(\_\_m256i a, \_\_m256i b);

VPCMPEQW \_\_mmask16 \_\_mm256\_mask\_cmpeq\_epi16\_mask(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPCMPEQW \_\_mmask8 \_\_mm\_cmpeq\_epi16\_mask(\_\_m128i a, \_\_m128i b);  
VPCMPEQW \_\_mmask8 \_\_mm\_mask\_cmpeq\_epi16\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
PCMPEQB \_\_m128i \_\_mm\_cmpeq\_epi8 (\_\_m128i a, \_\_m128i b)  
PCMPEQW \_\_m128i \_\_mm\_cmpeq\_epi16 (\_\_m128i a, \_\_m128i b)  
PCMPEQD \_\_m128i \_\_mm\_cmpeq\_epi32 (\_\_m128i a, \_\_m128i b)  
PCMPEQQ \_\_m128i \_\_mm\_cmpeq\_epi64(\_\_m128i a, \_\_m128i b);  
PCMPEQB \_\_m256i \_\_mm256\_cmpeq\_epi8 (\_\_m256i a, \_\_m256i b)  
PCMPEQW \_\_m256i \_\_mm256\_cmpeq\_epi16 (\_\_m256i a, \_\_m256i b)  
PCMPEQD \_\_m256i \_\_mm256\_cmpeq\_epi32 (\_\_m256i a, \_\_m256i b)  
PCMPEQQ \_\_m256i \_\_mm256\_cmpeq\_epi64(\_\_m256i a, \_\_m256i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.  
EVEX-encoded VPCMPEQD/Q, see Exceptions Type E4.  
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.



## PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ—Compare Packed Integers for Greater Than

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|-----------------------------|--------------------------|---|
| 66 0F 64 /r<br>PCMPGTB xmm1, xmm2/m128                                      | RM        | V/V                         | SSE2                     | Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.   |
| 66 0F 65 /r<br>PCMPGTW xmm1, xmm2/m128                                      | RM        | V/V                         | SSE2                     | Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.   |
| 66 0F 66 /r<br>PCMPGTD xmm1, xmm2/m128                                      | RM        | V/V                         | SSE2                     | Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.   |
| 66 0F 38 37 /r<br>PCMPGTQ xmm1, xmm2/m128                                   | RM        | V/V                         | SSE4_2                   | Compare packed qwords in xmm2/m128 and xmm1 for greater than.   |
| VEX.NDS.128.66.0F.WIG 64 /r<br>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<br>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<br>VPCMPGTD xmm1, xmm2, xmm3/m128               | RVM       | V/V                         | AVX                      | Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.   |
| VEX.NDS.128.66.0F38.WIG 37 /r<br>VPCMPGTQ xmm1, xmm2, xmm3/m128             | RVM       | V/V                         | AVX                      | Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.  |
| VEX.NDS.256.66.0F.WIG 64 /r<br>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<br>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<br>VPCMPGTD ymm1, ymm2, ymm3/m256               | RVM       | V/V                         | AVX2                     | Compare packed signed doubleword integers in ymm2 and ymm3/m256 for greater than.   |
| VEX.NDS.256.66.0F38.WIG 37 /r<br>VPCMPGTQ ymm1, ymm2, ymm3/m256             | RVM       | V/V                         | AVX2                     | Compare packed signed qwords in ymm2 and ymm3/m256 for greater than.  |
| EVEX.NDS.128.66.0F.W0 66 /r<br>VPCMPGTD k1 {k2}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.256.66.0F.W0 66 /r<br>VPCMPGTD k1 {k2}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.512.66.0F.W0 66 /r<br>VPCMPGTD k1 {k2}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                         | AVX512F                  | Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.                                   |

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F38.W1 37 /r<br>VPCMPGTQ k1 {k2}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.256.66.0F38.W1 37 /r<br>VPCMPGTQ k1 {k2}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.512.66.0F38.W1 37 /r<br>VPCMPGTQ k1 {k2}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask. |
| EVEX.NDS.128.66.0F.WIG 64 /r<br>VPCMPGTB k1 {k2}, xmm2,<br>xmm3/m128          | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.   |
| EVEX.NDS.256.66.0F.WIG 64 /r<br>VPCMPGTB k1 {k2}, ymm2, ymm3/m256             | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.   |
| EVEX.NDS.512.66.0F.WIG 64 /r<br>VPCMPGTB k1 {k2}, zmm2, zmm3/m512             | FVM       | V/V                          | AVX512BW                 | Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.   |
| EVEX.NDS.128.66.0F.WIG 65 /r<br>VPCMPGTW k1 {k2}, xmm2,<br>xmm3/m128          | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.   |
| EVEX.NDS.256.66.0F.WIG 65 /r<br>VPCMPGTW k1 {k2}, ymm2,<br>ymm3/m256          | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.   |
| EVEX.NDS.512.66.0F.WIG 65 /r<br>VPCMPGTW k1 {k2}, zmm2,<br>zmm3/m512          | FVM       | V/V                          | AVX512BW                 | Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.   |

### 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD signed compare for the greater value of the packed byte, word, doubleword, or quadword integers in the first source operand and the second source operand. If a data element in the first source operand is greater than the corresponding data element in the second 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 first and second source operands; the PCMPGTW instruction compares the corresponding signed word integers in the first and second source operands; the PCMPGTD instruction compares the corresponding signed doubleword integers in the first and second source operands, and the PCMPGTQ instruction compares the corresponding signed qword integers in the first and second source operands.

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). 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.

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 (MAX\_VL-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 (MAX\_VL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

## Operation

### COMPARE\_BYTES\_GREATER (SRC1, SRC2)

```
IF SRC1[7:0] > SRC2[7:0]
THEN DEST[7:0] ← FFH;
ELSE DEST[7:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 \*)

```
IF SRC1[127:120] > SRC2[127:120]
THEN DEST[127:120] ← FFH;
ELSE DEST[127:120] ← 0; FI;
```

### COMPARE\_WORDS\_GREATER (SRC1, SRC2)

```
IF SRC1[15:0] > SRC2[15:0]
THEN DEST[15:0] ← FFFFH;
ELSE DEST[15:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 \*)

```
IF SRC1[127:112] > SRC2[127:112]
THEN DEST[127:112] ← FFFFH;
ELSE DEST[127:112] ← 0; FI;
```

### COMPARE\_DWORDS\_GREATER (SRC1, SRC2)

```
IF SRC1[31:0] > SRC2[31:0]
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 \*)

```
IF SRC1[127:96] > SRC2[127:96]
THEN DEST[127:96] ← FFFFFFFFH;
ELSE DEST[127:96] ← 0; FI;
```

**COMPARE\_QWORDS\_GREATER (SRC1, SRC2)**

```

IF SRC1[63:0] > SRC2[63:0]
  THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] > SRC2[127:64]
  THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] ← 0; FI;

```

**VPCMPGTB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k2[j] OR \*no writemask\*

THEN

/\* signed comparison \*/

CMP ← SRC1[i+7:i] &gt; SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPGTB (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_BYTES\_GREATER(SRC1[127:0], SRC2[127:0])

DEST[255:128] ← COMPARE\_BYTES\_GREATER(SRC1[255:128], SRC2[255:128])

DEST[MAX\_VL-1:256] ← 0

**VPCMPGTB (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_BYTES\_GREATER(SRC1[127:0], SRC2[127:0])

DEST[MAX\_VL-1:128] ← 0

**PCMPGTB (128-bit Legacy SSE version)**

DEST[127:0] ← COMPARE\_BYTES\_GREATER(DEST[127:0], SRC[127:0])

DEST[MAX\_VL-1:128] (Unmodified)

**VPCMPGTW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

/\* signed comparison \*/

CMP ← SRC1[i+15:i] &gt; SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPGTW (VEX.256 encoded version)**

```
DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAX_VL-1:256] ← 0
```

**VPCMPGTW (VEX.128 encoded version)**

```
DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[MAX_VL-1:128] ← 0
```

**PCMPGTW (128-bit Legacy SSE version)**

```
DEST[127:0] ← COMPARE_WORDS_GREATER(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

**VPCMPGTD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (8, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
        ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
      FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPCMPGTD (VEX.256 encoded version)**

```
DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAX_VL-1:256] ← 0
```

**VPCMPGTD (VEX.128 encoded version)**

```
DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[MAX_VL-1:128] ← 0
```

**PCMPGTD (128-bit Legacy SSE version)**

```
DEST[127:0] ← COMPARE_DWORDS_GREATER(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

**VPCMPGTQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

/\* signed comparison \*/

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+63:i] &gt; SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] &gt; SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPGTQ (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_QWORDS\_GREATER(SRC1[127:0], SRC2[127:0])

DEST[255:128] ← COMPARE\_QWORDS\_GREATER(SRC1[255:128], SRC2[255:128])

DEST[MAX\_VL-1:256] ← 0

**VPCMPGTQ (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_QWORDS\_GREATER(SRC1[127:0], SRC2[127:0])

DEST[MAX\_VL-1:128] ← 0

**PCMPGTQ (128-bit Legacy SSE version)**

DEST[127:0] ← COMPARE\_QWORDS\_GREATER(DEST[127:0], SRC2[127:0])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPGTB \_\_mmask64 \_\_mm512\_cmpgt\_epi8\_mask(\_\_m512i a, \_\_m512i b);

VPCMPGTB \_\_mmask64 \_\_mm512\_mask\_cmpgt\_epi8\_mask(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPCMPGTB \_\_mmask32 \_\_mm256\_cmpgt\_epi8\_mask(\_\_m256i a, \_\_m256i b);

VPCMPGTB \_\_mmask32 \_\_mm256\_mask\_cmpgt\_epi8\_mask(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPCMPGTB \_\_mmask16 \_\_mm\_cmpgt\_epi8\_mask(\_\_m128i a, \_\_m128i b);

VPCMPGTB \_\_mmask16 \_\_mm\_mask\_cmpgt\_epi8\_mask(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPCMPGTD \_\_mmask16 \_\_mm512\_cmpgt\_epi32\_mask(\_\_m512i a, \_\_m512i b);

VPCMPGTD \_\_mmask16 \_\_mm512\_mask\_cmpgt\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPCMPGTQ \_\_mmask8 \_\_mm512\_cmpgt\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPCMPGTQ \_\_mmask8 \_\_mm512\_mask\_cmpgt\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPCMPGTD \_\_mmask8 \_\_mm256\_cmpgt\_epi32\_mask(\_\_m256i a, \_\_m256i b);

VPCMPGTD \_\_mmask8 \_\_mm256\_mask\_cmpgt\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPCMPGTQ \_\_mmask8 \_\_mm256\_cmpgt\_epi64\_mask(\_\_m256i a, \_\_m256i b);

VPCMPGTQ \_\_mmask8 \_\_mm256\_mask\_cmpgt\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPCMPGTD \_\_mmask8 \_\_mm\_cmpgt\_epi32\_mask(\_\_m128i a, \_\_m128i b);

VPCMPGTD \_\_mmask8 \_\_mm\_mask\_cmpgt\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPCMPGTQ \_\_mmask8 \_\_mm\_cmpgt\_epi64\_mask(\_\_m128i a, \_\_m128i b);

VPCMPGTQ \_\_mmask8 \_\_mm\_mask\_cmpgt\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPCMPGTW \_\_mmask32 \_\_mm512\_cmpgt\_epi16\_mask(\_\_m512i a, \_\_m512i b);

VPCMPGTW \_\_mmask32 \_\_mm512\_mask\_cmpgt\_epi16\_mask(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPCMPGTW \_\_mmask16 \_\_mm256\_cmpgt\_epi16\_mask(\_\_m256i a, \_\_m256i b);

VPCMPGTW \_\_mmask16 \_\_mm256\_mask\_cmpgt\_epi16\_mask(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPCMPGTW \_\_mmask8 \_\_mm\_cmpgt\_epi16\_mask(\_\_m128i a, \_\_m128i b);  
 VPCMPGTW \_\_mmask8 \_\_mm\_mask\_cmpgt\_epi16\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PCMPGTB \_\_m128i \_\_mm\_cmpgt\_epi8 (\_\_m128i a, \_\_m128i b)  
 PCMPGTW \_\_m128i \_\_mm\_cmpgt\_epi16 (\_\_m128i a, \_\_m128i b)  
 PCMPGTD \_\_m128i \_\_mm\_cmpgt\_epi32 (\_\_m128i a, \_\_m128i b)  
 PCMPGTQ \_\_m128i \_\_mm\_cmpgt\_epi64(\_\_m128i a, \_\_m128i b);  
 PCMPGTB \_\_m256i \_\_mm256\_cmpgt\_epi8 (\_\_m256i a, \_\_m256i b)  
 PCMPGTW \_\_m256i \_\_mm256\_cmpgt\_epi16 (\_\_m256i a, \_\_m256i b)  
 PCMPGTD \_\_m256i \_\_mm256\_cmpgt\_epi32 (\_\_m256i a, \_\_m256i b)  
 PCMPGTQ \_\_m256i \_\_mm256\_cmpgt\_epi64(\_\_m256i a, \_\_m256i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTD/Q, see Exceptions Type E4.

EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

## VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F3A.W0 3F /r ib<br>VPCMPB k1 {k2}, xmm2,<br>xmm3/m128, imm8  | FVM       | V/V                         | AVX512VL<br>AVX512BW     | Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.256.66.0F3A.W0 3F /r ib<br>VPCMPB k1 {k2}, ymm2,<br>ymm3/m256, imm8  | FVM       | V/V                         | AVX512VL<br>AVX512BW     | Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.512.66.0F3A.W0 3F /r ib<br>VPCMPB k1 {k2}, zmm2,<br>zmm3/m512, imm8  | FVM       | V/V                         | AVX512BW                 | Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.128.66.0F3A.W0 3E /r ib<br>VPCMPUB k1 {k2}, xmm2,<br>xmm3/m128, imm8 | FVM       | V/V                         | AVX512VL<br>AVX512BW     | Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.256.66.0F3A.W0 3E /r ib<br>VPCMPUB k1 {k2}, ymm2,<br>ymm3/m256, imm8 | FVM       | V/V                         | AVX512VL<br>AVX512BW     | Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.512.66.0F3A.W0 3E /r ib<br>VPCMPUB k1 {k2}, zmm2,<br>zmm3/m512, imm8 | FVM       | V/V                         | AVX512BW                 | Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (w) | vvvv (r)  | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.



Table 5-17. Pseudo-Op and VPCMP\* Implementation

| Pseudo-Op                          | PCMPM Implementation              |
|------------------------------------|-----------------------------------|
| VPCMPEQ* <i>reg1, reg2, reg3</i>   | VPCMP* <i>reg1, reg2, reg3, 0</i> |
| VPCMPLT* <i>reg1, reg2, reg3</i>   | VPCMP* <i>reg1, reg2, reg3, 1</i> |
| VPCMPLE* <i>reg1, reg2, reg3</i>   | VPCMP* <i>reg1, reg2, reg3, 2</i> |
| VPCMPNEQ* <i>reg1, reg2, reg3</i>  | VPCMP* <i>reg1, reg2, reg3, 4</i> |
| VPPCMPNLT* <i>reg1, reg2, reg3</i> | VPCMP* <i>reg1, reg2, reg3, 5</i> |
| VPCMPNLE* <i>reg1, reg2, reg3</i>  | VPCMP* <i>reg1, reg2, reg3, 6</i> |

**Operation**

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← FALSE;
- 4: OP ← NEQ;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← TRUE;

ESAC;

**VPCMPB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k2[j] OR \*no writemask\*

THEN

CMP ← SRC1[i+7:i] OP SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k2[j] OR \*no writemask\*

THEN

CMP ← SRC1[i+7:i] OP SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPB __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __mmask16 _mm_mask_cmp_epi8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64 _mm512_cmp_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F3A.W0 1F /r ib<br>VPCMPD k1 {k2}, xmm2,<br>xmm3/m128/m32bcst, imm8  | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                     |
| EVEX.NDS.256.66.0F3A.W0 1F /r ib<br>VPCMPD k1 {k2}, ymm2,<br>ymm3/m256/m32bcst, imm8  | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                     |
| EVEX.NDS.512.66.0F3A.W0 1F /r ib<br>VPCMPD k1 {k2}, zmm2,<br>zmm3/m512/m32bcst, imm8  | FV        | V/V                         | AVX512F                  | Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.   |
| EVEX.NDS.128.66.0F3A.W0 1E /r ib<br>VPCMPUD k1 {k2}, xmm2,<br>xmm3/m128/m32bcst, imm8 | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                   |
| EVEX.NDS.256.66.0F3A.W0 1E /r ib<br>VPCMPUD k1 {k2}, ymm2,<br>ymm3/m256/m32bcst, imm8 | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                   |
| EVEX.NDS.512.66.0F3A.W0 1E /r ib<br>VPCMPUD k1 {k2}, zmm2,<br>zmm3/m512/m32bcst, imm8 | FV        | V/V                         | AVX512F                  | Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;  
 1: OP ← LT;  
 2: OP ← LE;  
 3: OP ← FALSE;  
 4: OP ← NEQ;  
 5: OP ← NLT;  
 6: OP ← NLE;  
 7: OP ← TRUE;

ESAC;

**VPCMPD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPD \_\_mmask16 \_\_mm512\_cmp\_epi32\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPD \_\_mmask16 \_\_mm512\_mask\_cmp\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b, int imm);

VPCMPD \_\_mmask16 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPCMPD \_\_mmask16 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPCMPUD \_\_mmask16 \_\_mm512\_cmp\_epu32\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPUD \_\_mmask16 \_\_mm512\_mask\_cmp\_epu32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b, int imm);  
 VPCMPUD \_\_mmask16 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask( \_\_m512i a, \_\_m512i b);  
 VPCMPUD \_\_mmask16 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPCMPD \_\_mmask8 \_\_mm256\_cmp\_epi32\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm256\_mask\_cmp\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPD \_\_mmask8 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPUD \_\_mmask8 \_\_mm256\_cmp\_epu32\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm256\_mask\_cmp\_epu32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPUD \_\_mmask8 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPD \_\_mmask8 \_\_mm\_cmp\_epi32\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm\_mask\_cmp\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPD \_\_mmask8 \_\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPCMPUD \_\_mmask8 \_\_mm\_cmp\_epu32\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm\_mask\_cmp\_epu32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPUD \_\_mmask8 \_\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F3A.W1 1F /r ib<br>VPCMPQ k1 {k2}, xmm2,<br>xmm3/m128/m64bcst, imm8  | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.256.66.0F3A.W1 1F /r ib<br>VPCMPQ k1 {k2}, ymm2,<br>ymm3/m256/m64bcst, imm8  | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.512.66.0F3A.W1 1F /r ib<br>VPCMPQ k1 {k2}, zmm2,<br>zmm3/m512/m64bcst, imm8  | FV        | V/V                          | AVX512F                  | Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.128.66.0F3A.W1 1E /r ib<br>VPCMPUQ k1 {k2}, xmm2,<br>xmm3/m128/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.256.66.0F3A.W1 1E /r ib<br>VPCMPUQ k1 {k2}, ymm2,<br>ymm3/m256/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.512.66.0F3A.W1 1E /r ib<br>VPCMPUQ k1 {k2}, zmm2,<br>zmm3/m512/m64bcst, imm8 | FV        | V/V                          | AVX512F                  | Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

**Operation**

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← FALSE;
- 4: OP ← NEQ;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← TRUE;

ESAC;

**VPCMPQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k2[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

        ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k2[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

        ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPQ \_\_mmask8 \_mm512\_cmp\_epi64\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPQ \_\_mmask8 \_mm512\_mask\_cmp\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, int imm);

VPCMPQ \_\_mmask8 \_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask( \_\_m512i a, \_\_m512i b);

VPCMPQ \_\_mmask8 \_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPCMPUQ \_\_mmask8 \_mm512\_cmp\_epu64\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPUQ \_\_mmask8 \_\_mm512\_mask\_cmp\_epu64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, int imm);  
 VPCMPUQ \_\_mmask8 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask( \_\_m512i a, \_\_m512i b);  
 VPCMPUQ \_\_mmask8 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPCMPQ \_\_mmask8 \_\_mm256\_mask\_cmp\_epi64\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPQ \_\_mmask8 \_\_mm256\_mask\_cmp\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPQ \_\_mmask8 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPQ \_\_mmask8 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPUQ \_\_mmask8 \_\_mm256\_cmp\_epu64\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUQ \_\_mmask8 \_\_mm256\_mask\_cmp\_epu64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUQ \_\_mmask8 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPUQ \_\_mmask8 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPQ \_\_mmask8 \_\_mm\_cmp\_epi64\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPQ \_\_mmask8 \_\_mm\_mask\_cmp\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPQ \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPQ \_\_mmask8 \_\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPCMPUQ \_\_mmask8 \_\_mm\_cmp\_epu64\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUQ \_\_mmask8 \_\_mm\_mask\_cmp\_epu64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUQ \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPUQ \_\_mmask8 \_\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.



## VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F3A.W1 3F /r ib<br>VPCMPW k1 {k2}, xmm2,<br>xmm3/m128, imm8  | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.256.66.0F3A.W1 3F /r ib<br>VPCMPW k1 {k2}, ymm2,<br>ymm3/m256, imm8  | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.512.66.0F3A.W1 3F /r ib<br>VPCMPW k1 {k2}, zmm2,<br>zmm3/m512, imm8  | FVM       | V/V                          | AVX512BW                 | Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.NDS.128.66.0F3A.W1 3E /r ib<br>VPCMPUW k1 {k2}, xmm2,<br>xmm3/m128, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.256.66.0F3A.W1 3E /r ib<br>VPCMPUW k1 {k2}, ymm2,<br>ymm3/m256, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.512.66.0F3A.W1 3E /r ib<br>VPCMPUW k1 {k2}, zmm2,<br>zmm3/m512, imm8 | FVM       | V/V                          | AVX512BW                 | Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (w) | vvvv (r)  | ModRM:r/m (r) | NA        |

#### Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;  
 1: OP ← LT;  
 2: OP ← LE;  
 3: OP ← FALSE;  
 4: OP ← NEQ;  
 5: OP ← NLT;  
 6: OP ← NLE;  
 7: OP ← TRUE;

ESAC;

**VPCMPW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

ICMP ← SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

CMP ← SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPW \_\_mmask32 \_\_mm512\_cmp\_epi16\_mask( \_\_m512i a, \_\_m512i b, int cmp);

VPCMPW \_\_mmask32 \_\_mm512\_mask\_cmp\_epi16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b, int cmp);

VPCMPW \_\_mmask16 \_\_mm256\_cmp\_epi16\_mask( \_\_m256i a, \_\_m256i b, int cmp);

VPCMPW \_\_mmask16 \_\_mm256\_mask\_cmp\_epi16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b, int cmp);

VPCMPW \_\_mmask8 \_\_mm\_cmp\_epi16\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPW \_\_mmask8 \_\_mm\_mask\_cmp\_epi16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int cmp);

VPCMPW \_\_mmask32 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPCMPW \_\_mmask32 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b);

VPCMPW \_\_mmask16 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_m256i a, \_\_m256i b);

VPCMPW \_\_mmask16 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b);

VPCMPW \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_m128i a, \_\_m128i b);

VPCMPW \_\_mmask8 \_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b);  
 VPCMPUW \_\_mmask32 \_mm512\_cmp\_epu16\_mask( \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask32 \_mm512\_mask\_cmp\_epu16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_cmp\_epu16\_mask( \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_mask\_cmp\_epu16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_cmp\_epu16\_mask( \_\_m128i a, \_\_m128i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_mask\_cmp\_epu16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int cmp);  
 VPCMPUW \_\_mmask32 \_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask32 \_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_m128i a, \_\_m128i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int cmp);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 8B /r<br>VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1. |
| EVEX.256.66.0F38.W0 8B /r<br>VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1. |
| EVEX.512.66.0F38.W0 8B /r<br>VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2 | T1S       | V/V                          | AVX512F                  | Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPCOMPRESSD (EVEX encoded versions) store form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no controlmask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+31:i]

      k ← k + SIZE

  FI;

ENDFOR;

**VPCOMPRESSD (EVEX encoded versions) reg-reg form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+31:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSD \_\_m512i \_\_mm512\_mask\_compress\_epi32(\_\_m512i s, \_\_mmask16 c, \_\_m512i a);

VPCOMPRESSD \_\_m512i \_\_mm512\_maskz\_compress\_epi32(\_\_mmask16 c, \_\_m512i a);

VPCOMPRESSD void \_\_mm512\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask16 c, \_\_m512i s);

VPCOMPRESSD \_\_m256i \_\_mm256\_mask\_compress\_epi32(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);

VPCOMPRESSD \_\_m256i \_\_mm256\_maskz\_compress\_epi32(\_\_mmask8 c, \_\_m256i a);

VPCOMPRESSD void \_\_mm256\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask8 c, \_\_m256i s);

VPCOMPRESSD \_\_m128i \_\_mm\_mask\_compress\_epi32(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);

VPCOMPRESSD \_\_m128i \_\_mm\_maskz\_compress\_epi32(\_\_mmask8 c, \_\_m128i a);

VPCOMPRESSD void \_\_mm\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask8 c, \_\_m128i s);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 8B /r<br>VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1. |
| EVEX.256.66.0F38.W1 8B /r<br>VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1. |
| EVEX.512.66.0F38.W1 8B /r<br>VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2 | T1S       | V/V                          | AVX512F                  | Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPCOMPRESSQ (EVEX encoded versions) store form

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no controlmask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+63:i]

      k ← k + SIZE

  FI;

ENFOR

**VPCOMPRESSQ (EVEX encoded versions) reg-reg form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no controlmask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+63:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSQ \_\_m512i \_\_mm512\_mask\_compress\_epi64(\_\_m512i s, \_\_mmask8 c, \_\_m512i a);

VPCOMPRESSQ \_\_m512i \_\_mm512\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m512i a);

VPCOMPRESSQ void \_\_mm512\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m512i s);

VPCOMPRESSQ \_\_m256i \_\_mm256\_mask\_compress\_epi64(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);

VPCOMPRESSQ \_\_m256i \_\_mm256\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m256i a);

VPCOMPRESSQ void \_\_mm256\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m256i s);

VPCOMPRESSQ \_\_m128i \_\_mm\_mask\_compress\_epi64(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);

VPCOMPRESSQ \_\_m128i \_\_mm\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m128i a);

VPCOMPRESSQ void \_\_mm\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m128i s);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 C4 /r<br>VPCONFLICTD xmm1 {k1}{z},<br>xmm2/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate double-word values in<br>xmm2/m128/m32bcst using writemask k1. |
| EVEX.256.66.0F38.W0 C4 /r<br>VPCONFLICTD ymm1 {k1}{z},<br>ymm2/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate double-word values in<br>ymm2/m256/m32bcst using writemask k1. |
| EVEX.512.66.0F38.W0 C4 /r<br>VPCONFLICTD zmm1 {k1}{z},<br>zmm2/m512/m32bcst | FV        | V/V                          | AVX512CD                 | Detect duplicate double-word values in<br>zmm2/m512/m32bcst using writemask k1. |
| EVEX.128.66.0F38.W1 C4 /r<br>VPCONFLICTQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate quad-word values in<br>xmm2/m128/m64bcst using writemask k1.   |
| EVEX.256.66.0F38.W1 C4 /r<br>VPCONFLICTQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate quad-word values in<br>ymm2/m256/m64bcst using writemask k1.   |
| EVEX.512.66.0F38.W1 C4 /r<br>VPCONFLICTQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst | FV        | V/V                          | AVX512CD                 | Detect duplicate quad-word values in<br>zmm2/m512/m64bcst using writemask k1.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.



**Operation****VPCONFLICTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\* THEN

FOR k ← 0 TO j-1

m ← k\*32

IF ((SRC[i+31:i] = SRC[m+31:m])) THEN

DEST[i+k] ← 1

ELSE

DEST[i+k] ← 0

FI

ENDFOR

DEST[i+31:i+j] ← 0

ELSE

IF \*merging-masking\* THEN

\*DEST[i+31:i] remains unchanged\*

ELSE

DEST[i+31:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPCONFLICTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\* THEN

FOR k ← 0 TO j-1

m ← k\*64

IF ((SRC[i+63:i] = SRC[m+63:m])) THEN

DEST[i+k] ← 1

ELSE

DEST[i+k] ← 0

FI

ENDFOR

DEST[i+63:i+j] ← 0

ELSE

IF \*merging-masking\* THEN

\*DEST[i+63:i] remains unchanged\*

ELSE

DEST[i+63:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCONFLICTD \_\_m512i \_mm512\_conflict\_epi32(\_\_m512i a);  
 VPCONFLICTD \_\_m512i \_mm512\_mask\_conflict\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a);  
 VPCONFLICTD \_\_m512i \_mm512\_maskz\_conflict\_epi32(\_\_mmask16 m, \_\_m512i a);  
 VPCONFLICTQ \_\_m512i \_mm512\_conflict\_epi64(\_\_m512i a);  
 VPCONFLICTQ \_\_m512i \_mm512\_mask\_conflict\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a);  
 VPCONFLICTQ \_\_m512i \_mm512\_maskz\_conflict\_epi64(\_\_mmask8 m, \_\_m512i a);  
 VPCONFLICTD \_\_m256i \_mm256\_conflict\_epi32(\_\_m256i a);  
 VPCONFLICTD \_\_m256i \_mm256\_mask\_conflict\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTD \_\_m256i \_mm256\_maskz\_conflict\_epi32(\_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTQ \_\_m256i \_mm256\_conflict\_epi64(\_\_m256i a);  
 VPCONFLICTQ \_\_m256i \_mm256\_mask\_conflict\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTQ \_\_m256i \_mm256\_maskz\_conflict\_epi64(\_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTD \_\_m128i \_mm\_conflict\_epi32(\_\_m128i a);  
 VPCONFLICTD \_\_m128i \_mm\_mask\_conflict\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
 VPCONFLICTD \_\_m128i \_mm\_maskz\_conflict\_epi32(\_\_mmask8 m, \_\_m128i a);  
 VPCONFLICTQ \_\_m128i \_mm\_conflict\_epi64(\_\_m128i a);  
 VPCONFLICTQ \_\_m128i \_mm\_mask\_conflict\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
 VPCONFLICTQ \_\_m128i \_mm\_maskz\_conflict\_epi64(\_\_mmask8 m, \_\_m128i a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.

## VPERMB—Permute Packed Bytes Elements

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature Flag  | Description  |
|--|-----------|------------------------------|------------------------|--|
| EVEX.NDS.128.66.0F38.W0 8D /r<br>VPERMB xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512VBMI | Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F38.W0 8D /r<br>VPERMB ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512VBMI | Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 8D /r<br>VPERMB zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512VBMI             | Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

### Operation

#### VPERMB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

n ← 3;

ELSE IF VL = 256:

n ← 4;

ELSE IF VL = 512:

n ← 5;

FI;

FOR j ← 0 TO KL-1:

id ← SRC1[j\*8 + n : j\*8]; // location of the source byte

IF k1[j] OR \*no writemask\* THEN

DEST[j\*8 + 7 : j\*8] ← SRC2[id\*8 + 7: id\*8];

ELSE IF zeroing-masking THEN

DEST[j\*8 + 7 : j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7 : j\*8] remains unchanged\*

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

### Intel C/C++ Compiler Intrinsic Equivalent

```
VPERMB __m512i __mm512_permutexvar_epi8( __m512i idx, __m512i a);  
VPERMB __m512i __mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);  
VPERMB __m512i __mm512_maskz_permutexvar_epi8( __mmask64 k, __m512i idx, __m512i a);  
VPERMB __m256i __mm256_permutexvar_epi8( __m256i idx, __m256i a);  
VPERMB __m256i __mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);  
VPERMB __m256i __mm256_maskz_permutexvar_epi8( __mmask32 k, __m256i idx, __m256i a);  
VPERMB __m128i __mm_permutexvar_epi8( __m128i idx, __m128i a);  
VPERMB __m128i __mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);  
VPERMB __m128i __mm_maskz_permutexvar_epi8( __mmask16 k, __m128i idx, __m128i a);
```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4NF.nb.

## VPERMD/VPERMW—Permute Packed Doublewords/Words Elements

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.NDS.256.66.0F38.W0 36 /r<br>VPERMD ymm1, ymm2, ymm3/m256                     | RVM        | V/V                          | AVX2                     | Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.                            |
| EVEX.NDS.256.66.0F38.W0 36 /r<br>VPERMD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 36 /r<br>VPERMD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1. |
| EVEX.NDS.128.66.0F38.W1 8D /r<br>VPERMW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.       |
| EVEX.NDS.256.66.0F38.W1 8D /r<br>VPERMW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.       |
| EVEX.NDS.512.66.0F38.W1 8D /r<br>VPERMW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | FVM        | V/V                          | AVX512BW                 | Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.       |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| RVM   | ModRM:reg (w) | VEX.vvvv  | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |
| FVM   | ModRM:reg (w) | VEX.vvvv  | ModRM:r/m (r) | NA        |

### Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

**Operation****VPERMD (EVEX encoded versions)**

```

(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n ← 2; FI;
IF VL = 512 THEN n ← 3; FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  id ← 32*SRC1[i:i+n]
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] ← SRC2[31:0];
        ELSE DEST[i+31:i] ← SRC2[id+31:id];
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**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];
DEST[MAX_VL-1:256] ← 0

```

**VPERMW (EVEX encoded versions)**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n ← 2; FI;
IF VL = 256 THEN n ← 3; FI;
IF VL = 512 THEN n ← 4; FI;
FOR j ← 0 TO KL-1
  i ← j * 16
  id ← 16*SRC1[i:i+n]
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC2[id+15:id]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+15:i] ← 0
        FI
      FI;
    ENDFOR
  FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMD \_\_m512i \_\_mm512\_permutexvar\_epi32( \_\_m512i idx, \_\_m512i a);  
 VPERMD \_\_m512i \_\_mm512\_mask\_permutexvar\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i idx, \_\_m512i a);  
 VPERMD \_\_m512i \_\_mm512\_maskz\_permutexvar\_epi32(\_\_mmask16 k, \_\_m512i idx, \_\_m512i a);  
 VPERMD \_\_m256i \_\_mm256\_permutexvar\_epi32( \_\_m256i idx, \_\_m256i a);  
 VPERMD \_\_m256i \_\_mm256\_mask\_permutexvar\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i idx, \_\_m256i a);  
 VPERMD \_\_m256i \_\_mm256\_maskz\_permutexvar\_epi32(\_\_mmask8 k, \_\_m256i idx, \_\_m256i a);  
 VPERMW \_\_m512i \_\_mm512\_permutexvar\_epi16( \_\_m512i idx, \_\_m512i a);  
 VPERMW \_\_m512i \_\_mm512\_mask\_permutexvar\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i idx, \_\_m512i a);  
 VPERMW \_\_m512i \_\_mm512\_maskz\_permutexvar\_epi16(\_\_mmask32 k, \_\_m512i idx, \_\_m512i a);  
 VPERMW \_\_m256i \_\_mm256\_permutexvar\_epi16( \_\_m256i idx, \_\_m256i a);  
 VPERMW \_\_m256i \_\_mm256\_mask\_permutexvar\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i idx, \_\_m256i a);  
 VPERMW \_\_m256i \_\_mm256\_maskz\_permutexvar\_epi16(\_\_mmask16 k, \_\_m256i idx, \_\_m256i a);  
 VPERMW \_\_m128i \_\_mm\_permutexvar\_epi16( \_\_m128i idx, \_\_m128i a);  
 VPERMW \_\_m128i \_\_mm\_mask\_permutexvar\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i idx, \_\_m128i a);  
 VPERMW \_\_m128i \_\_mm\_maskz\_permutexvar\_epi16(\_\_mmask8 k, \_\_m128i idx, \_\_m128i a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPERMD, see Exceptions Type E4NF.

EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD                    If VEX.L = 0.  
                       If EVEX.L'L = 0 for VPERMD.

### VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag  | Description   |
|--|-----------|------------------------------|------------------------|---|
| EVEX.DDS.128.66.0F38.W0 75 /r<br>VPERMI2B xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512VBMI | Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W0 75 /r<br>VPERMI2B ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512VBMI | Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W0 75 /r<br>VPERMI2B zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512VBMI             | Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1. |

#### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2 | Operand 3     | Operand 4 |
|-------|------------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

#### Description

Permutes byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.



**Operation****VPERMI2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id: j\*8];

IF k1[j] OR \*no writemask\*:

DEST[j\*8 + 7: j\*8] ← TMP\_DEST[j\*8+id+1]? SRC2[off+7:off] : SRC1[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7: j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7: j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMI2B \_\_m512i \_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2B \_\_m512i \_mm512\_mask2\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_mmask64 k, \_\_m512i b);

VPERMI2B \_\_m512i \_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2B \_\_m256i \_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2B \_\_m256i \_mm256\_mask2\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_mmask32 k, \_\_m256i b);

VPERMI2B \_\_m256i \_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2B \_\_m128i \_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMI2B \_\_m128i \_mm\_mask2\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_mmask16 k, \_\_m128i b);

VPERMI2B \_\_m128i \_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.DDS.128.66.0F38.W1 75 /r<br>VPERMI2W xmm1 {k1}{z}, xmm2,<br>xmm3/m128          | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.                      |
| EVEX.DDS.256.66.0F38.W1 75 /r<br>VPERMI2W ymm1 {k1}{z}, ymm2,<br>ymm3/m256          | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.                      |
| EVEX.DDS.512.66.0F38.W1 75 /r<br>VPERMI2W zmm1 {k1}{z}, zmm2,<br>zmm3/m512          | FVM        | V/V                          | AVX512BW                 | Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.                      |
| EVEX.DDS.128.66.0F38.W0 76 /r<br>VPERMI2D xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst  | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.               |
| EVEX.DDS.256.66.0F38.W0 76 /r<br>VPERMI2D ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst  | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.               |
| EVEX.DDS.512.66.0F38.W0 76 /r<br>VPERMI2D zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst  | FV         | V/V                          | AVX512F                  | Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.               |
| EVEX.DDS.128.66.0F38.W1 76 /r<br>VPERMI2Q xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst  | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.                 |
| EVEX.DDS.256.66.0F38.W1 76 /r<br>VPERMI2Q ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst  | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.                 |
| EVEX.DDS.512.66.0F38.W1 76 /r<br>VPERMI2Q zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst  | FV         | V/V                          | AVX512F                  | Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.                 |
| EVEX.DDS.128.66.0F38.W0 77 /r<br>VPERMI2PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W0 77 /r<br>VPERMI2PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W0 77 /r<br>VPERMI2PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1. |

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.DDS.128.66.0F38.W1 77 /r<br>VPERMI2PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W1 77 /r<br>VPERMI2PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W1 77 /r<br>VPERMI2PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| FVM   | ModRM:reg (r,w)  | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

#### Description

Permutates 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMI2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id ← 2

FI;

IF VL = 256

id ← 3

FI;

IF VL = 512

id ← 4

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 16

off ← 16 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] = TMP\_DEST[i+id+1] ? SRC2[off+15:off]  
: SRC1[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMI2D/VPERMI2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

id ← 1

FI;

IF VL = 256

id ← 2

FI;

IF VL = 512

id ← 3

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 32

off ← 32 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← TMP\_DEST[i+id+1] ? SRC2[31:0]  
: SRC1[off+31:off]

ELSE

DEST[i+31:i] ← TMP\_DEST[i+id+1] ? SRC2[off+31:off]  
: SRC1[off+31:off]

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPERMI2Q/VPERMI2PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

id ← 0

FI;

IF VL = 256

id ← 1

FI;

IF VL = 512

id ← 2

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 64

off ← 64 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← TMP\_DEST[i+id+1] ? SRC2[63:0]

: SRC1[off+63:off]

ELSE

DEST[i+63:i] ← TMP\_DEST[i+id+1] ? SRC2[off+63:off]

: SRC1[off+63:off]

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMI2D \_\_m512i \_\_mm512\_permutex2var\_epi32(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2D \_\_m512i \_\_mm512\_mask\_permutex2var\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i idx, \_\_m512i b);

VPERMI2D \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi32(\_\_m512i a, \_\_m512i idx, \_\_mmask16 k, \_\_m512i b);

VPERMI2D \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI \_\_m256i \_\_mm256\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2D \_\_m256i \_\_mm256\_mask\_permutex2var\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);

VPERMI2D \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMI2D \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2D \_\_m128i \_\_mm\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2D \_\_m128i \_\_mm\_mask\_permutex2var\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2D \_\_m128i \_\_mm\_mask2\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2D \_\_m128i \_\_mm\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2PD \_\_m512d \_\_mm512\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_\_mm512\_mask\_permutex2var\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_\_mm512\_mask2\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_mmask8 k, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_\_mm512\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_mask\_permutex2var\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_mask2\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_mmask8 k, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m128d \_\_mm\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_\_mm\_mask\_permutex2var\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_\_mm\_mask2\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_mmask8 k, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_\_mm\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PS \_\_m512 \_\_mm512\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_\_mm512\_mask\_permutex2var\_ps(\_\_m512 a, \_\_mmask16 k, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_\_mm512\_mask2\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_mmask16 k, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_\_mm512\_maskz\_permutex2var\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_mask\_permutex2var\_ps(\_\_m256 a, \_\_mmask8 k, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_mask2\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_mmask8 k, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m128 \_\_mm\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_\_mm\_mask\_permutex2var\_ps(\_\_m128 a, \_\_mmask8 k, \_\_m128i idx, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_\_mm\_mask2\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_mmask8 k, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_\_mm\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMI2Q \_\_m512i \_\_mm512\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_\_mm512\_mask\_permutex2var\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_mmask8 k, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_mask\_permutex2var\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m128i \_\_mm\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_\_mm\_mask\_permutex2var\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_\_mm\_mask2\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_\_mm\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2W \_\_m512i \_\_mm512\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_mask\_permutex2var\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_mmask32 k, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m256i \_\_mm256\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_mask\_permutex2var\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_mmask16 k, \_\_m256i b);

VPERMI2W \_\_m256i \_mm256\_maskz\_permutex2var\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2W \_\_m128i \_mm\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMI2W \_\_m128i \_mm\_mask\_permutex2var\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);

VPERMI2W \_\_m128i \_mm\_mask2\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);

VPERMI2W \_\_m128i \_mm\_maskz\_permutex2var\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VPERMI2D/Q/PS/PD: See Exceptions Type E4NF.

VPERMI2W: See Exceptions Type E4NF.nb.

## VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag  | Description   |
|--|-----------|------------------------------|------------------------|---|
| EVEX.DDS.128.66.0F38.W0 7D /r<br>VPERMT2B xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512VBMI | Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F38.W0 7D /r<br>VPERMT2B ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512VBMI | Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 7D /r<br>VPERMT2B zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512VBMI             | Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2 | Operand 3     | Operand 4 |
|-------|------------------|-----------|---------------|-----------|
| FVM   | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.



**Operation****VPERMT2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id; j\*8];

IF k1[j] OR \*no writemask\*:

DEST[j\*8 + 7: j\*8] ← SRC1[j\*8+id+1]? SRC2[off+7:off] : TMP\_DEST[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7: j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7: j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMT2B \_\_m512i \_\_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i \_\_mm512\_mask\_permutex2var\_epi8(\_\_m512i a, \_\_mmask64 k, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m256i \_\_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i \_\_mm256\_mask\_permutex2var\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m128i \_\_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i \_\_mm\_mask\_permutex2var\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i \_\_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.DDS.128.66.0F38.W1 7D /r<br>VPERMT2W xmm1 {k1}{z}, xmm2,<br>xmm3/m128          | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.                      |
| EVEX.DDS.256.66.0F38.W1 7D /r<br>VPERMT2W ymm1 {k1}{z}, ymm2,<br>ymm3/m256          | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.                      |
| EVEX.DDS.512.66.0F38.W1 7D /r<br>VPERMT2W zmm1 {k1}{z}, zmm2,<br>zmm3/m512          | FVM       | V/V                          | AVX512BW                 | Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.                      |
| EVEX.DDS.128.66.0F38.W0 7E /r<br>VPERMT2D xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst  | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.               |
| EVEX.DDS.256.66.0F38.W0 7E /r<br>VPERMT2D ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst  | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.               |
| EVEX.DDS.512.66.0F38.W0 7E /r<br>VPERMT2D zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst  | FV        | V/V                          | AVX512F                  | Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.               |
| EVEX.DDS.128.66.0F38.W1 7E /r<br>VPERMT2Q xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst  | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.                 |
| EVEX.DDS.256.66.0F38.W1 7E /r<br>VPERMT2Q ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst  | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.                 |
| EVEX.DDS.512.66.0F38.W1 7E /r<br>VPERMT2Q zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst  | FV        | V/V                          | AVX512F                  | Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.                 |
| EVEX.DDS.128.66.0F38.W0 7F /r<br>VPERMT2PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W0 7F /r<br>VPERMT2PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W0 7F /r<br>VPERMT2PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1. |

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.DDS.128.66.0F38.W1 7F /r<br>VPERMT2PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W1 7F /r<br>VPERMT2PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W1 7F /r<br>VPERMT2PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| FVM   | ModRM:reg (r,w)  | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

#### Description

Permutates 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMT2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id ← 2

FI;

IF VL = 256

id ← 3

FI;

IF VL = 512

id ← 4

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 16

off ← 16 \* SRC1[j+id:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] = SRC1[i+id+1] ? SRC2[off+15:off]

: TMP\_DEST[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMT2D/VPERMT2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

id ← 1

FI;

IF VL = 256

id ← 2

FI;

IF VL = 512

id ← 3

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 32

off ← 32 \* SRC1[j+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[31:0]

: TMP\_DEST[off+31:off]

ELSE

DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[off+31:off]

: TMP\_DEST[off+31:off]

```

    FI
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                           ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPERMT2Q/VPERMT2PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

id ← 0

FI;

IF VL = 256

id ← 1

FI;

IF VL = 512

id ← 2

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 64

off ← 64 \* SRC1[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[63:0]

: TMP\_DEST[off+63:off]

ELSE

DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[off+63:off]

: TMP\_DEST[off+63:off]

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMT2D \_\_m512i \_\_mm512\_permutex2var\_epi32(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2D \_\_m512i \_\_mm512\_mask\_permutex2var\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i idx, \_\_m512i b);

VPERMT2D \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi32(\_\_m512i a, \_\_m512i idx, \_\_mmask16 k, \_\_m512i b);

VPERMT2D \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2D \_\_m256i \_\_mm256\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2D \_\_m256i \_\_mm256\_mask\_permutex2var\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);

VPERMT2D \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMT2D \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2D \_\_m128i \_\_mm\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2D \_\_m128i \_\_mm\_mask\_permutex2var\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMT2D \_\_m128i \_\_mm\_mask2\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMT2D \_\_m128i \_\_mm\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2PD \_\_m512d \_\_mm512\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMT2PD \_\_m512d \_\_mm512\_mask\_permutex2var\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512i idx, \_\_m512d b);  
 VPERMT2PD \_\_m512d \_\_mm512\_mask2\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_mmask8 k, \_\_m512d b);  
 VPERMT2PD \_\_m512d \_\_mm512\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMT2PD \_\_m256d \_\_mm256\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMT2PD \_\_m256d \_\_mm256\_mask\_permutex2var\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256i idx, \_\_m256d b);  
 VPERMT2PD \_\_m256d \_\_mm256\_mask2\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_mmask8 k, \_\_m256d b);  
 VPERMT2PD \_\_m256d \_\_mm256\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMT2PD \_\_m128d \_\_mm\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMT2PD \_\_m128d \_\_mm\_mask\_permutex2var\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128i idx, \_\_m128d b);  
 VPERMT2PD \_\_m128d \_\_mm\_mask2\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_mmask8 k, \_\_m128d b);  
 VPERMT2PD \_\_m128d \_\_mm\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMT2PS \_\_m512 \_\_mm512\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMT2PS \_\_m512 \_\_mm512\_mask\_permutex2var\_ps(\_\_m512 a, \_\_mmask16 k, \_\_m512i idx, \_\_m512 b);  
 VPERMT2PS \_\_m512 \_\_mm512\_mask2\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_mmask16 k, \_\_m512 b);  
 VPERMT2PS \_\_m512 \_\_mm512\_maskz\_permutex2var\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_mask\_permutex2var\_ps(\_\_m256 a, \_\_mmask8 k, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_mask2\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_mmask8 k, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m128 \_\_mm\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_mask\_permutex2var\_ps(\_\_m128 a, \_\_mmask8 k, \_\_m128i idx, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_mask2\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_mmask8 k, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMT2Q \_\_m512i \_\_mm512\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_mask\_permutex2var\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_mmask8 k, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_mask\_permutex2var\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m128i \_\_mm\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_mask\_permutex2var\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_mask2\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m512i \_\_mm512\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_mask\_permutex2var\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_mmask32 k, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m256i \_\_mm256\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_mask\_permutex2var\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_mmask16 k, \_\_m256i b);

VPERMT2W \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2W \_\_m128i \_\_mm\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMT2W \_\_m128i \_\_mm\_mask\_permutex2var\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);

VPERMT2W \_\_m128i \_\_mm\_mask2\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);

VPERMT2W \_\_m128i \_\_mm\_maskz\_permutex2var\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VPERMT2D/Q/PS/PD: See Exceptions Type E4NF.

VPERMT2W: See Exceptions Type E4NF.nb.

## VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W0 0D /r<br>VPERMILPD xmm1, xmm2, xmm3/m128                     | RVM     | V/V                          | AVX                      | Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.                                |
| VEX.NDS.256.66.0F38.W0 0D /r<br>VPERMILPD ymm1, ymm2, ymm3/m256                     | RVM     | V/V                          | AVX                      | Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.                                |
| EVEX.NDS.128.66.0F38.W1 0D /r<br>VPERMILPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.  |
| EVEX.NDS.256.66.0F38.W1 0D /r<br>VPERMILPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.  |
| EVEX.NDS.512.66.0F38.W1 0D /r<br>VPERMILPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV-RVM  | V/V                          | AVX512F                  | Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.  |
| VEX.128.66.0F3A.W0 05 /r ib<br>VPERMILPD xmm1, xmm2/m128, imm8                      | RM      | V/V                          | AVX                      | Permute double-precision floating-point values in xmm2/m128 using controls from imm8.   |
| VEX.256.66.0F3A.W0 05 /r ib<br>VPERMILPD ymm1, ymm2/m256, imm8                      | RM      | V/V                          | AVX                      | Permute double-precision floating-point values in ymm2/m256 using controls from imm8.   |
| EVEX.128.66.0F3A.W1 05 /r ib<br>VPERMILPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8  | FV-RM   | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F3A.W1 05 /r ib<br>VPERMILPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8  | FV-RM   | V/V                          | AVX512VL<br>AVX512F      | Permute double-precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F3A.W1 05 /r ib<br>VPERMILPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8  | FV-RM   | V/V                          | AVX512F                  | Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1. |

### 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        |
| RM     | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| FV-RVM | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV-RM  | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |



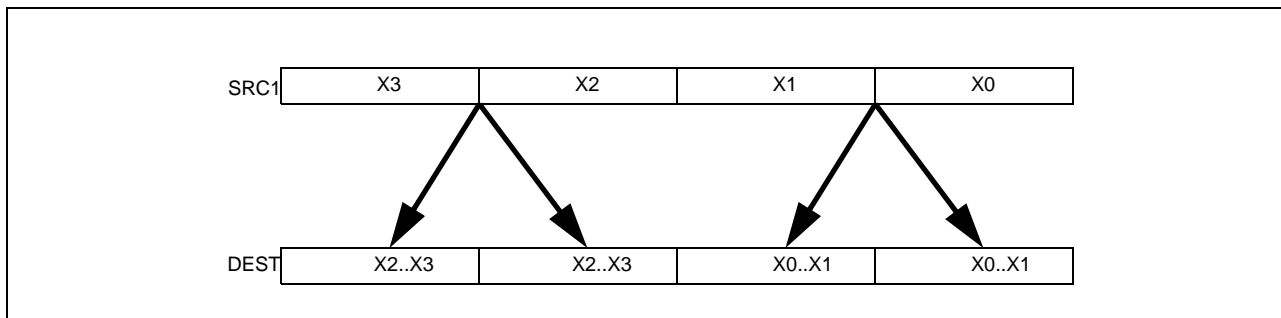
**Description**

(variable control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

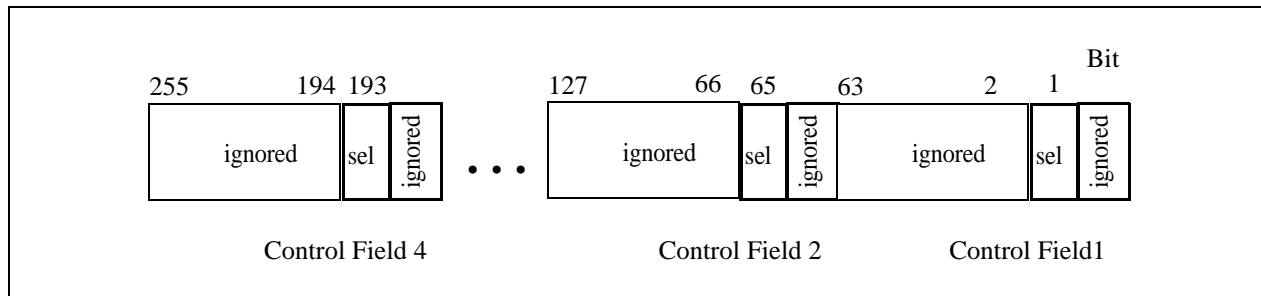
The control bits are located at bit 0 of each quadword element (see Figure 5-31). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.



**Figure 5-30. VPERMILPD Operation**

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.



**Figure 5-31. VPERMILPD Shuffle Control**

(immediate control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

**Operation****VPERMILPD (EVEX immediate versions)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN TMP\_SRC1[i+63:i] ← SRC1[63:0];

ELSE TMP\_SRC1[i+63:i] ← SRC1[i+63:i];

FI;

ENDFOR;

IF (imm8[0] = 0) THEN TMP\_DEST[63:0] ← SRC1[63:0]; FI;

IF (imm8[0] = 1) THEN TMP\_DEST[63:0] ← TMP\_SRC1[127:64]; FI;

IF (imm8[1] = 0) THEN TMP\_DEST[127:64] ← TMP\_SRC1[63:0]; FI;

IF (imm8[1] = 1) THEN TMP\_DEST[127:64] ← TMP\_SRC1[127:64]; FI;

IF VL ≥ 256

IF (imm8[2] = 0) THEN TMP\_DEST[191:128] ← TMP\_SRC1[191:128]; FI;

IF (imm8[2] = 1) THEN TMP\_DEST[191:128] ← TMP\_SRC1[255:192]; FI;

IF (imm8[3] = 0) THEN TMP\_DEST[255:192] ← TMP\_SRC1[191:128]; FI;

IF (imm8[3] = 1) THEN TMP\_DEST[255:192] ← TMP\_SRC1[255:192]; FI;

FI;

IF VL ≥ 512

IF (imm8[4] = 0) THEN TMP\_DEST[319:256] ← TMP\_SRC1[319:256]; FI;

IF (imm8[4] = 1) THEN TMP\_DEST[319:256] ← TMP\_SRC1[383:320]; FI;

IF (imm8[5] = 0) THEN TMP\_DEST[383:320] ← TMP\_SRC1[319:256]; FI;

IF (imm8[5] = 1) THEN TMP\_DEST[383:320] ← TMP\_SRC1[383:320]; FI;

IF (imm8[6] = 0) THEN TMP\_DEST[447:384] ← TMP\_SRC1[447:384]; FI;

IF (imm8[6] = 1) THEN TMP\_DEST[447:384] ← TMP\_SRC1[511:448]; FI;

IF (imm8[7] = 0) THEN TMP\_DEST[511:448] ← TMP\_SRC1[447:384]; FI;

IF (imm8[7] = 1) THEN TMP\_DEST[511:448] ← TMP\_SRC1[511:448]; FI;

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\*   ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE   ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**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]

DEST[MAX\_VL-1:256] ← 0

**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[MAX_VL-1:128] ← 0

```

**VPERMILPD (EVEX variable versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 64
```

```
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
```

```
  FI;
```

```
ENDFOR;
```

```

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] ← SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] ← SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] ← SRC1[127:64]; FI;
IF VL >= 256
  IF (TMP_SRC2[129] = 0) THEN TMP_DEST[191:128] ← SRC1[191:128]; FI;
  IF (TMP_SRC2[129] = 1) THEN TMP_DEST[191:128] ← SRC1[255:192]; FI;
  IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] ← SRC1[191:128]; FI;
  IF (TMP_SRC2[193] = 1) THEN TMP_DEST[255:192] ← SRC1[255:192]; FI;

```

```
FI;
```

```
IF VL >= 512
```

```

  IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] ← SRC1[319:256]; FI;
  IF (TMP_SRC2[257] = 1) THEN TMP_DEST[319:256] ← SRC1[383:320]; FI;
  IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] ← SRC1[319:256]; FI;
  IF (TMP_SRC2[321] = 1) THEN TMP_DEST[383:320] ← SRC1[383:320]; FI;
  IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] ← SRC1[447:384]; FI;
  IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] ← SRC1[511:448]; FI;
  IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] ← SRC1[447:384]; FI;
  IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] ← SRC1[511:448]; FI;

```

```
FI;
```

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 64
```

```
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
```

```
  ELSE
```

```
    IF *merging-masking* ; merging-masking
```

```
      THEN *DEST[i+63:i] remains unchanged*
```

```
    ELSE ; zeroing-masking
```

```
      DEST[i+63:i] ← 0
```

```
  FI
```

```
  FI;
```

```
ENDFOR
```

```
DEST[MAX_VL-1:VL] ← 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]
DEST[MAX_VL-1:256] ← 0

```

**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[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPD __m512d __mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d __mm512_mask_permute_pd( __m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d __mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d __mm256_mask_permute_pd( __m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d __mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d __mm_mask_permute_pd( __m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d __mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m512d __mm512_permutevar_pd( __m512i i, __m512d a);
VPERMILPD __m512d __mm512_mask_permutevar_pd( __m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d __mm512_maskz_permutevar_pd( __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d __mm256_mask_permutevar_pd( __m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m256d __mm256_maskz_permutevar_pd( __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m128d __mm_mask_permutevar_pd( __m128d s, __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d __mm_maskz_permutevar_pd( __mmask8 k, __m128d i, __m128d a);
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**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|---------|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 0C /r<br>VPERMILPS xmm1, xmm2, xmm3/m128                     | RVM     | V/V                          | AVX                      | Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.                             |
| VEX.128.66.0F3A.W0 04 /r ib<br>VPERMILPS xmm1, xmm2/m128, imm8                      | RM      | V/V                          | AVX                      | Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.                             |
| VEX.NDS.256.66.0F3A.W0 0C /r<br>VPERMILPS ymm1, ymm2, ymm3/m256                     | RVM     | V/V                          | AVX                      | Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.                             |
| VEX.256.66.0F3A.W0 04 /r ib<br>VPERMILPS ymm1, ymm2/m256, imm8                      | RM      | V/V                          | AVX                      | Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.                             |
| EVEX.NDS.128.66.0F3A.W0 0C /r<br>VPERMILPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.  |
| EVEX.NDS.256.66.0F3A.W0 0C /r<br>VPERMILPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.  |
| EVEX.NDS.512.66.0F3A.W0 0C /r<br>VPERMILPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV-RVM  | V/V                          | AVX512F                  | Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.  |
| EVEX.128.66.0F3A.W0 04 /r ib<br>VPERMILPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8  | FV-RM   | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F3A.W0 04 /r ib<br>VPERMILPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8  | FV-RM   | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F3A.W0 04 /r ib<br>VPERMILPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8  | FV-RM   | V/V                          | AVX512F                  | Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1. |

### 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        |
| RM     | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |
| FV-RVM | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV-RM  | ModRM:reg (w) | ModRM:r/m (r) | NA            | NA        |

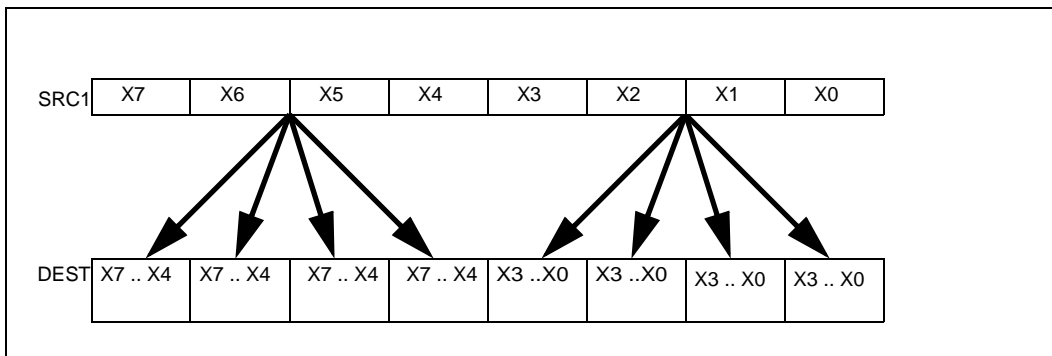
**Description**

(variable control version)

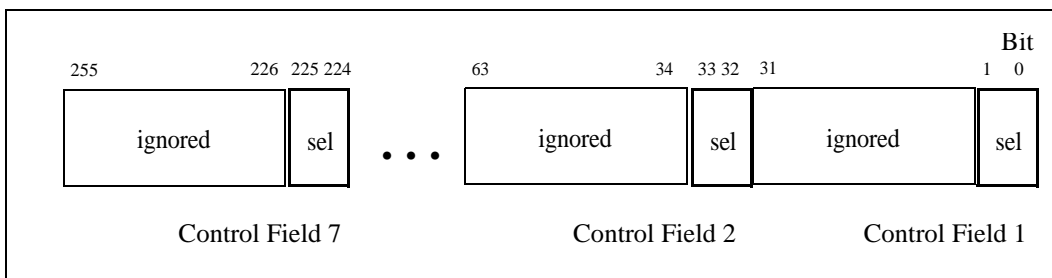
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-33). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.



**Figure 5-32. VPERMILPS Operation**



**Figure 5-33. VPERMILPS Shuffle Control**

(immediate control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are 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 (EVEX immediate versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

    THEN TMP\_SRC1[i+31:i] ← SRC1[31:0];

    ELSE TMP\_SRC1[i+31:i] ← SRC1[i+31:i];

  FI;

ENDFOR;

TMP\_DEST[31:0] ← Select4(TMP\_SRC1[127:0], imm8[1:0]);

TMP\_DEST[63:32] ← Select4(TMP\_SRC1[127:0], imm8[3:2]);

TMP\_DEST[95:64] ← Select4(TMP\_SRC1[127:0], imm8[5:4]);

TMP\_DEST[127:96] ← Select4(TMP\_SRC1[127:0], imm8[7:6]); FI;

IF VL >= 256

  TMP\_DEST[159:128] ← Select4(TMP\_SRC1[255:128], imm8[1:0]); FI;

  TMP\_DEST[191:160] ← Select4(TMP\_SRC1[255:128], imm8[3:2]); FI;

  TMP\_DEST[223:192] ← Select4(TMP\_SRC1[255:128], imm8[5:4]); FI;

  TMP\_DEST[255:224] ← Select4(TMP\_SRC1[255:128], imm8[7:6]); FI;

FI;

IF VL >= 512

  TMP\_DEST[287:256] ← Select4(TMP\_SRC1[383:256], imm8[1:0]); FI;

  TMP\_DEST[319:288] ← Select4(TMP\_SRC1[383:256], imm8[3:2]); FI;

  TMP\_DEST[351:320] ← Select4(TMP\_SRC1[383:256], imm8[5:4]); FI;

  TMP\_DEST[383:352] ← Select4(TMP\_SRC1[383:256], imm8[7:6]); FI;

  TMP\_DEST[415:384] ← Select4(TMP\_SRC1[511:384], imm8[1:0]); FI;

  TMP\_DEST[447:416] ← Select4(TMP\_SRC1[511:384], imm8[3:2]); FI;

  TMP\_DEST[479:448] ← Select4(TMP\_SRC1[511:384], imm8[5:4]); FI;

  TMP\_DEST[511:480] ← Select4(TMP\_SRC1[511:384], imm8[7:6]); FI;

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

    ELSE

      IF \*merging-masking\*

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE DEST[i+31:i] ← 0 ;zeroing-masking

    FI;

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**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[MAX_VL-1:128] ← 0

```

**VPERMILPS (EVEX variable versions)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

    THEN TMP\_SRC2[i+31:i] ← SRC2[31:0];

    ELSE TMP\_SRC2[i+31:i] ← SRC2[i+31:i];

  FI;

ENDFOR;

TMP\_DEST[31:0] ← Select4(SRC1[127:0], TMP\_SRC2[1:0]);

TMP\_DEST[63:32] ← Select4(SRC1[127:0], TMP\_SRC2[33:32]);

TMP\_DEST[95:64] ← Select4(SRC1[127:0], TMP\_SRC2[65:64]);

TMP\_DEST[127:96] ← Select4(SRC1[127:0], TMP\_SRC2[97:96]);

IF VL >= 256

  TMP\_DEST[159:128] ← Select4(SRC1[255:128], TMP\_SRC2[129:128]);

  TMP\_DEST[191:160] ← Select4(SRC1[255:128], TMP\_SRC2[161:160]);

  TMP\_DEST[223:192] ← Select4(SRC1[255:128], TMP\_SRC2[193:192]);

  TMP\_DEST[255:224] ← Select4(SRC1[255:128], TMP\_SRC2[225:224]);

FI;

IF VL >= 512

  TMP\_DEST[287:256] ← Select4(SRC1[383:256], TMP\_SRC2[257:256]);

  TMP\_DEST[319:288] ← Select4(SRC1[383:256], TMP\_SRC2[289:288]);

  TMP\_DEST[351:320] ← Select4(SRC1[383:256], TMP\_SRC2[321:320]);

  TMP\_DEST[383:352] ← Select4(SRC1[383:256], TMP\_SRC2[353:352]);

  TMP\_DEST[415:384] ← Select4(SRC1[511:384], TMP\_SRC2[385:384]);

  TMP\_DEST[447:416] ← Select4(SRC1[511:384], TMP\_SRC2[417:416]);

  TMP\_DEST[479:448] ← Select4(SRC1[511:384], TMP\_SRC2[449:448]);

  TMP\_DEST[511:480] ← Select4(SRC1[511:384], TMP\_SRC2[481:480]);

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

  ELSE

    IF \*merging-masking\*

      THEN \*DEST[i+31:i] remains unchanged\*

      ELSE DEST[i+31:i] ← 0 ;zeroing-masking



```

    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 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]);
DEST[MAX_VL-1:256] ← 0

```

**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[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPS __m512 __mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __mm512_mask_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __mm256_mask_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __mm_mask_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 __mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __mm512_mask_permutevar_ps( __m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 __mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 __mm256_mask_permutevar_ps( __m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 __mm_mask_permutevar_ps( __m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __mm_maskz_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
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**

Non-EVEX-encoded instruction, see Exceptions Type 4;

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMPD—Permute Double-Precision Floating-Point Elements

| Opcode/<br>Instruction  | Op / En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|-----------------------------|--------------------------|---|
| VEX.256.66.0F3A.W1 01 /r ib<br>VPERMPD ymm1, ymm2/m256, imm8                      | RMI     | V/V                         | AVX2                     | Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.                                 |
| EVEX.256.66.0F3A.W1 01 /r ib<br>VPERMPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8  | FV-RM   | V/V                         | AVX512VL<br>AVX512F      | Permute double-precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 01 /r ib<br>VPERMPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8  | FV-RMI  | V/V                         | AVX512F                  | Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F38.W1 16 /r<br>VPERMPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV-RVM  | V/V                         | AVX512VL<br>AVX512F      | Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F38.W1 16 /r<br>VPERMPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV-RVM  | V/V                         | AVX512F                  | Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|--------|---------------|---------------|---------------|-----------|
| RMI    | ModRM:reg (w) | ModRM:r/m (r) | Imm8          | NA        |
| FV-RMI | ModRM:reg (w) | ModRM:r/m (r) | Imm8          | NA        |
| FV-RVM | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation****VPERMPD (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

  IF (EVEX.b = 1) AND (SRC \*is memory\*)  
    THEN TMP\_SRC[i+63:i] ← SRC[63:0];  
    ELSE TMP\_SRC[i+63:i] ← SRC[i+63:i];

FI;

ENDFOR;

TMP\_DEST[63:0] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

IF VL &gt;= 512

TMP\_DEST[319:256] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[383:320] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[447:384] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[511:448] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

  IF k1[j] OR \*no writemask\*  
    THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]  
    ELSE      IF \*merging-masking\*                   ;merging-masking  
        THEN \*DEST[i+63:i] remains unchanged\*

ELSE                                   ;zeroing-masking

DEST[i+63:i] ← 0                   ;zeroing-masking

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMPD (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)  
    THEN TMP\_SRC2[i+63:i] ← SRC2[63:0];  
    ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i];

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[63:0] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[65:64] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[129:128] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[193:192] \* 64))[63:0];

FI;

IF VL = 512

TMP\_DEST[63:0] ← (TMP\_SRC2[511:0] &gt;&gt; (SRC1[2:0] \* 64))[63:0];

```

TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[j+63:i]
  ELSE
    IF *merging-masking* ;merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ;zeroing-masking
      DEST[i+63:i] ← 0 ;zeroing-masking
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**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];
DEST[MAX_VL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMPD __m512d __mm512_permutex_pd( __m512d a, int imm);
VPERMPD __m512d __mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __mm512_maskz_permutex_pd( __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __mm512_permutexvar_pd( __m512i i, __m512d a);
VPERMPD __m512d __mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d __mm512_maskz_permutexvar_pd( __mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d __mm256_permutex_epi64( __m256d a, int imm);
VPERMPD __m256d __mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __mm256_maskz_permutex_epi64( __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __mm256_permutexvar_epi64( __m256i i, __m256d a);
VPERMPD __m256d __mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d __mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i i, __m256d a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

```

#UD          If VEX.L = 0.
              If VEX.vvvv != 1111B.

```

EVEX-encoded instruction, see Exceptions Type E4NF.

```

#UD          If encoded with EVEX.128.
              If EVEX.vvvv != 1111B and with imm8.

```

## VPERMPS—Permute Single-Precision Floating-Point Elements

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| VEX.256.66.0F38.W0 16 /r<br>VPERMPS ymm1, ymm2,<br>ymm3/m256                      | RVM        | V/V                          | AVX2                     | Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.                                  |
| EVEX.NDS.256.66.0F38.W0 16 /r<br>VPERMPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1. |
| EVEX.NDS.512.66.0F38.W0 16 /r<br>VPERMPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.   |

### 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        |
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VPERMPS (EVEX forms)

(KL, VL) (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

    THEN TMP\_SRC2[i+31:i] ← SRC2[31:0];

    ELSE TMP\_SRC2[i+31:i] ← SRC2[i+31:i];

  FI;

ENDFOR;

IF VL = 256

  TMP\_DEST[31:0] ← (TMP\_SRC2[255:0] >> (SRC1[2:0] \* 32))[31:0];

  TMP\_DEST[63:32] ← (TMP\_SRC2[255:0] >> (SRC1[34:32] \* 32))[31:0];

  TMP\_DEST[95:64] ← (TMP\_SRC2[255:0] >> (SRC1[66:64] \* 32))[31:0];

  TMP\_DEST[127:96] ← (TMP\_SRC2[255:0] >> (SRC1[98:96] \* 32))[31:0];

  TMP\_DEST[159:128] ← (TMP\_SRC2[255:0] >> (SRC1[130:128] \* 32))[31:0];

```

    TMP_DEST[191:160] ← (TMP_SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
FI;
IF VL = 512
    TMP_DEST[31:0] ← (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] ← (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] ← (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] ← (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] ← (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] ← (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] ← (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] ← (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] ← (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] ← (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] ← (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] ← (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] ← (TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] ← (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[j+31:j]
    ELSE
        IF *merging-masking* ;merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ;zeroing-masking
            DEST[i+31:i] ← 0 ;zeroing-masking
    FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**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];
DEST[MAX_VL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMPS __m512 __mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 __mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 __mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 __mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 __mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 __mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally  
#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

## VPERMQ—Qwords Element Permutation

| Opcode/<br>Instruction   | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|------------------------------|--------------------------|---|
| VEX.256.66.0F3A.W1 00 /r ib<br>VPERMQ ymm1, ymm2/m256, imm8                      | RMI     | V/V                          | AVX2                     | Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.         |
| EVEX.256.66.0F3A.W1 00 /r ib<br>VPERMQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8  | FV-RM   | V/V                          | AVX512VL<br>AVX512F      | Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1. |
| EVEX.512.66.0F3A.W1 00 /r ib<br>VPERMQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8  | FV-RMI  | V/V                          | AVX512F                  | Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1. |
| EVEX.NDS.256.66.0F38.W1 36 /r<br>VPERMQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1. |
| EVEX.NDS.512.66.0F38.W1 36 /r<br>VPERMQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV-RVM  | V/V                          | AVX512F                  | Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1. |

### Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|--------|---------------|---------------|---------------|-----------|
| RMI    | ModRM:reg (w) | ModRM:r/m (r) | Imm8          | NA        |
| FV-RMI | ModRM:reg (w) | ModRM:r/m (r) | Imm8          | NA        |
| FV-RVM | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.



**Operation****VPERMQ (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC \*is memory\*)  
 THEN TMP\_SRC[i+63:i] ← SRC[63:0];  
 ELSE TMP\_SRC[i+63:i] ← SRC[i+63:i];

FI;

ENDFOR;

TMP\_DEST[63:0] ← (TMP\_SRC[255:0] >> (IMM8[1:0] \* 64))[63:0];  
 TMP\_DEST[127:64] ← (TMP\_SRC[255:0] >> (IMM8[3:2] \* 64))[63:0];  
 TMP\_DEST[191:128] ← (TMP\_SRC[255:0] >> (IMM8[5:4] \* 64))[63:0];  
 TMP\_DEST[255:192] ← (TMP\_SRC[255:0] >> (IMM8[7:6] \* 64))[63:0];

IF VL &gt;= 512

TMP\_DEST[319:256] ← (TMP\_SRC[511:256] >> (IMM8[1:0] \* 64))[63:0];  
 TMP\_DEST[383:320] ← (TMP\_SRC[511:256] >> (IMM8[3:2] \* 64))[63:0];  
 TMP\_DEST[447:384] ← (TMP\_SRC[511:256] >> (IMM8[5:4] \* 64))[63:0];  
 TMP\_DEST[511:448] ← (TMP\_SRC[511:256] >> (IMM8[7:6] \* 64))[63:0];

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]  
 ELSE

IF \*merging-masking\* ;merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ;zeroing-masking

DEST[i+63:i] ← 0 ;zeroing-masking

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMQ (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)  
 THEN TMP\_SRC2[i+63:i] ← SRC2[63:0];  
 ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i];

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[63:0] ← (TMP\_SRC2[255:0] >> (SRC1[1:0] \* 64))[63:0];  
 TMP\_DEST[127:64] ← (TMP\_SRC2[255:0] >> (SRC1[65:64] \* 64))[63:0];  
 TMP\_DEST[191:128] ← (TMP\_SRC2[255:0] >> (SRC1[129:128] \* 64))[63:0];  
 TMP\_DEST[255:192] ← (TMP\_SRC2[255:0] >> (SRC1[193:192] \* 64))[63:0];

FI;

IF VL = 512

TMP\_DEST[63:0] ← (TMP\_SRC2[511:0] >> (SRC1[2:0] \* 64))[63:0];  
 TMP\_DEST[127:64] ← (TMP\_SRC2[511:0] >> (SRC1[66:64] \* 64))[63:0];  
 TMP\_DEST[191:128] ← (TMP\_SRC2[511:0] >> (SRC1[130:128] \* 64))[63:0];  
 TMP\_DEST[255:192] ← (TMP\_SRC2[511:0] >> (SRC1[194:192] \* 64))[63:0];

```

TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ;merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ;zeroing-masking
      DEST[i+63:i] ← 0 ;zeroing-masking
  FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**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];
DEST[MAX_VL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMQ __m512i __mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i __mm512_mask_permutex_epi64( __m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i __mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i __mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i __mm512_mask_permutexvar_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i __mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i __mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i __mm256_mask_permutex_epi64( __m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i __mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i __mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i __mm256_mask_permutexvar_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i __mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

```

#UD          If VEX.L = 0.
             If VEX.vvvv != 1111B.

```

EVEX-encoded instruction, see Exceptions Type E4NF.

```

#UD          If encoded with EVEX.128.
             If EVEX.vvvv != 1111B and with imm8.

```

## VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 89 /r<br>VPEXPANDD xmm1 {k1}{z},<br>xmm2/m128 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 89 /r<br>VPEXPANDD ymm1 {k1}{z},<br>ymm2/m256 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 89 /r<br>VPEXPANDD zmm1 {k1}{z},<br>zmm2/m512 | T1S       | V/V                          | AVX512F                  | Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

$k \leftarrow 0$

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 32$

  IF  $k1[j]$  OR \*no writemask\*

    THEN

$DEST[i+31:i] \leftarrow SRC[k+31:k];$

$k \leftarrow k + 32$

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

    FI

  FI;

ENDFOR

$DEST[MAX\_VL-1:VL] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDD __m512i __mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VEXPANDD __m512i __mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VEXPANDD __m512i __mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VEXPANDD __m512i __mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VEXPANDD __m256i __mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VEXPANDD __m256i __mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VEXPANDD __m256i __mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VEXPANDD __m256i __mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VEXPANDD __m128i __mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VEXPANDD __m128i __mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VEXPANDD __m128i __mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VEXPANDD __m128i __mm_maskz_expand_epi32(__mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 89 /r<br>VPEXPANDQ xmm1 {k1}{z}, xmm2/m128 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 89 /r<br>VPEXPANDQ ymm1 {k1}{z}, ymm2/m256 | T1S       | V/V                          | AVX512VL<br>AVX512F      | Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 89 /r<br>VPEXPANDQ zmm1 {k1}{z}, zmm2/m512 | T1S       | V/V                          | AVX512F                  | Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+63:i] ← SRC[k+63:k];

      k ← k + 64

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

          THEN DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDQ __m512i __mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VEXPANDQ __m512i __mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m512i __mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VEXPANDQ __m512i __mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VEXPANDQ __m256i __mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VEXPANDQ __m256i __mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m256i __mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VEXPANDQ __m256i __mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VEXPANDQ __m128i __mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VEXPANDQ __m128i __mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m128i __mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VEXPANDQ __m128i __mm_maskz_expand_epi64(__mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## PEXTRB/PEXTRW/PEXTRD/PEXTRQ—Extract Integer

| Opcode/<br>Instruction                                       | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|------------------------------|--------------------------|---|
| 66 0F 3A 14 /r ib<br>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 r64/r32 is filled with zeros.  |
| 66 0F C5 /r ib<br>PEXTRW reg, xmm1, imm8                     | RMI     | V/V                          | SSE2                     | Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. The upper bits of r64/r32 is filled with zeros.                         |
| 66 0F 3A 15 /r ib<br>PEXTRW reg/m16, xmm2, imm8              | MRI     | V/V                          | SSE4_1                   | 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. |
| 66 0F 3A 16 /r ib<br>PEXTRD r32/m32, xmm2, imm8              | MRI     | V/V                          | SSE4_1                   | Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.  |
| 66 REX.W 0F 3A 16 /r ib<br>PEXTRQ r64/m64, xmm2, imm8        | MRI     | V/N.E.                       | SSE4_1                   | Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.  |
| VEX.128.66.0F3A 14 /r ib<br>VPEXTRB reg/m8, xmm2, imm8       | MRI     | V/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.128.66.0F C5 /r ib<br>VPEXTRW reg, xmm1, imm8            | RMI     | V/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 15 /r ib<br>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. |
| VEX.128.66.0F3A.W0 16 /r ib<br>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<br>VPEXTRQ r64/m64, xmm2, imm8   | MRI     | V/N.E.                       | AVX                      | Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.  |
| EVEX.128.66.0F3A.WIG 14 /r ib<br>VPEXTRB reg/m8, xmm2, imm8  | T1S-MRI | V/V                          | AVX512BW                 | Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.  |
| EVEX.128.66.0F.WIG C5 /r ib<br>VPEXTRW reg, xmm1, imm8       | RMI     | V/V                          | AVX512BW                 | Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros. |
| EVEX.128.66.0F3A.WIG 15 /r ib<br>VPEXTRW reg/m16, xmm2, imm8 | T1S-MRI | V/V                          | AVX512BW                 | Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros. |
| EVEX.128.66.0F3A.W0 16 /r ib<br>VPEXTRD r32/m32, xmm2, imm8  | T1S-MRI | V/V                          | AVX512DQ                 | Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.  |
| EVEX.128.66.0F3A.W1 16 /r ib<br>VPEXTRQ r64/m64, xmm2, imm8  | T1S-MRI | V/N.E. <sup>1</sup>          | AVX512DQ                 | Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.  |

## NOTES:

1. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

## Instruction Operand Encoding

| Op/En   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|---------|---------------|---------------|-----------|-----------|
| MRI     | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |
| RMI     | ModRM:reg (w) | ModRM:r/m(r)  | Imm8      | NA        |
| T1S-MRI | ModRM:r/m (w) | ModRM:reg (r) | Imm8      | NA        |

## Description

Extract a byte/word/dword/qword integer value from the source XMM register at a byte/word/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/word/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In 64-bit mode, if the destination operand is a register, default operand size is 64 bits. The bits above the least significant dword/word/byte data element are filled with zeros

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

## Operation

**(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 0FFFFFFFh);

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

**(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  $\leftarrow$  Imm8[3:0]DEST[7:0]  $\leftarrow$  ((Src  $\gg$  Src\_Offset\*8) AND 0FFh)DEST[63:8]  $\leftarrow$  ZERO\_FILL;

ELSE

SRC\_Offset  $\leftarrow$  Imm8[3:0];DEST[7:0]  $\leftarrow$  ((Src  $\gg$  Src\_Offset\*8) AND 0FFh);DEST[31:8]  $\leftarrow$  ZERO\_FILL;

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

PEXTRB int \_mm\_extract\_epi8 (\_\_m128i src, const int ndx);

PEXTRW int \_mm\_extract\_epi16 (\_\_m128i src, int ndx);

PEXTRD int \_mm\_extract\_epi32 (\_\_m128i src, const int ndx);

PEXTRQ \_\_int64 \_mm\_extract\_epi64 (\_\_m128i src, const int ndx);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD

If VEX.L = 1 or EVEX.L'L &gt; 0.

If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 44 /r<br>VPLZCNTD xmm1 {k1}{z},<br>xmm2/m128/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1. |
| EVEX.256.66.0F38.W0 44 /r<br>VPLZCNTD ymm1 {k1}{z},<br>ymm2/m256/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1. |
| EVEX.512.66.0F38.W0 44 /r<br>VPLZCNTD zmm1 {k1}{z},<br>zmm2/m512/m32bcst | FV        | V/V                         | AVX512CD                 | Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1. |
| EVEX.128.66.0F38.W1 44 /r<br>VPLZCNTQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst | FV        | V/V                         | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1. |
| EVEX.256.66.0F38.W1 44 /r<br>VPLZCNTQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst | FV        | V/V                         | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1. |
| EVEX.512.66.0F38.W1 44 /r<br>VPLZCNTQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst | FV        | V/V                         | AVX512CD                 | Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPLZCNTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\*

THEN

temp ← 32

DEST[i+31:i] ← 0

WHILE (temp &gt; 0) AND (SRC[i+temp-1] = 0)

DO

temp ← temp - 1

DEST[i+31:i] ← DEST[i+31:i] + 1

OD

ELSE

IF \*merging-masking\*

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPLZCNTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\*

THEN

temp ← 64

DEST[i+63:i] ← 0

WHILE (temp &gt; 0) AND (SRC[i+temp-1] = 0)

DO

temp ← temp - 1

DEST[i+63:i] ← DEST[i+63:i] + 1

OD

ELSE

IF \*merging-masking\*

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPLZCNTD \_\_m512i \_\_mm512\_lzcnt\_epi32(\_\_m512i a);

VPLZCNTD \_\_m512i \_\_mm512\_mask\_lzcnt\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a);

VPLZCNTD \_\_m512i \_\_mm512\_maskz\_lzcnt\_epi32(\_\_mmask16 m, \_\_m512i a);

VPLZCNTQ \_\_m512i \_\_mm512\_lzcnt\_epi64(\_\_m512i a);

VPLZCNTQ \_\_m512i \_\_mm512\_mask\_lzcnt\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a);

VPLZCNTQ \_\_m512i \_\_mm512\_maskz\_lzcnt\_epi64(\_\_mmask8 m, \_\_m512i a);

VPLZCNTD \_\_m256i \_\_mm256\_lzcnt\_epi32(\_\_m256i a);

VPLZCNTD \_\_m256i \_mm256\_mask\_lzcnt\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
VPLZCNTD \_\_m256i \_mm256\_maskz\_lzcnt\_epi32(\_\_mmask8 m, \_\_m256i a);  
VPLZCNTQ \_\_m256i \_mm256\_lzcnt\_epi64(\_\_m256i a);  
VPLZCNTQ \_\_m256i \_mm256\_mask\_lzcnt\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
VPLZCNTQ \_\_m256i \_mm256\_maskz\_lzcnt\_epi64(\_\_mmask8 m, \_\_m256i a);  
VPLZCNTD \_\_m128i \_mm\_lzcnt\_epi32(\_\_m128i a);  
VPLZCNTD \_\_m128i \_mm\_mask\_lzcnt\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
VPLZCNTD \_\_m128i \_mm\_maskz\_lzcnt\_epi32(\_\_mmask8 m, \_\_m128i a);  
VPLZCNTQ \_\_m128i \_mm\_lzcnt\_epi64(\_\_m128i a);  
VPLZCNTQ \_\_m128i \_mm\_mask\_lzcnt\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
VPLZCNTQ \_\_m128i \_mm\_maskz\_lzcnt\_epi64(\_\_mmask8 m, \_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## PMADDUBSW—Multiply and Add Packed Integers

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F 38 04 /r<br>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 04 /r<br>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 04 /r<br>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 xmm1.                    |
| EVEX.NDS.128.66.0F38.WIG 04 /r<br>VPMADDUBSW xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.WIG 04 /r<br>VPMADDUBSW ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.WIG 04 /r<br>VPMADDUBSW zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

PMADDUBSW multiplies vertically each unsigned byte of the first source operand with the corresponding signed byte of the second source 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 first source and second source 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.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

**Operation****VPMADDUBSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPMADDUBSW (VEX.256 encoded version)**

```

DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAX_VL-1:256] ← 0

```

**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[MAX_VL-1:128] ← 0

```

**PMADDUBSW (128-bit Legacy SSE version)**

```

DEST[15:0] ← SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]* DEST[119:112]);
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMADDUBSW __m512i __mm512_mddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i __mm512_mask_mddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i __mm512_maskz_mddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i __mm256_mask_mddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i __mm256_maskz_mddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i __mm128_mask_mddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i __mm128_maskz_mddubs_epi16(__mmask8 k, __m128i a, __m128i b);
(V)PMADDUBSW __m128i __mm_maddubs_epi16(__m128i a, __m128i b)
VPMADDUBSW __m256i __mm256_maddubs_epi16(__m256i a, __m256i b)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PMADDWD—Multiply and Add Packed Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| 66 0F F5 /r<br>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 F5 /r<br>VPMADDWD xmm1, xmm2,<br>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 F5 /r<br>VPMADDWD ymm1, ymm2,<br>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.                    |
| EVEX.NDS.128.66.0F.WIG F5 /r<br>VPMADDWD xmm1 {k1}{z},<br>xmm2, xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG F5 /r<br>VPMADDWD ymm1 {k1}{z},<br>ymm2, ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.WIG F5 /r<br>VPMADDWD zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source 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 second source and first source 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.

The PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

### Operation

#### VPMADDWD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

#### VPMADDWD (VEX.256 encoded version)

```

DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[MAX_VL-1:256] ← 0

```

#### 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[MAX_VL-1:128] ← 0

```

#### PMADDWD (128-bit Legacy SSE version)

```

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])
DEST[MAX_VL-1:128] (Unmodified)

```



**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADDWD \_\_m512i \_mm512\_mdd\_epi16(\_\_m512i a, \_\_m512i b);  
 VPMADDWD \_\_m512i \_mm512\_mask\_mdd\_epi16(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMADDWD \_\_m512i \_mm512\_maskz\_mdd\_epi16(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMADDWD \_\_m256i \_mm256\_mask\_mdd\_epi16(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMADDWD \_\_m256i \_mm256\_maskz\_mdd\_epi16(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMADDWD \_\_m128i \_mm\_mask\_mdd\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMADDWD \_\_m128i \_mm\_maskz\_mdd\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMADDWD \_\_m128i \_mm\_madd\_epi16 (\_\_m128i a, \_\_m128i b)  
 VPMADDWD \_\_m256i \_mm256\_madd\_epi16 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

**PINSRB/PINSRW/PINSRD/PINSRQ—Insert Integer**

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| 66 0F 3A 20 /r ib<br>PINSRB xmm1, r32/m8, imm8                        | RMI     | V/V                          | SSE4_1                   | Insert a byte integer value from r32/m8 into xmm1 at the byte offset in imm8.                       |
| 66 0F C4 /r ib<br>PINSRW xmm1, r32/m16, imm8                          | RMI     | V/V                          | SSE2                     | Insert a word integer value from r32/m16 into xmm1 at the word offset in imm8.                      |
| 66 0F 3A 22 /r ib<br>PINSRD xmm1, r32/m32, imm8                       | RMI     | V/V                          | SSE4_1                   | Insert a dword integer value from r32/m32 into xmm1 at the dword offset in imm8.                    |
| 66 REX.W 0F 3A 22 /r ib<br>PINSRQ xmm1, r64/m64, imm8                 | RMI     | V/N.E.                       | SSE4_1                   | Insert a qword integer value from r64/m64 into xmm1 at the qword offset in imm8.                    |
| VEX.NDS.128.66.0F3A 20 /r ib<br>VPINSRB xmm1, xmm2, r32/m8, imm8      | RVMI    | V/V                          | AVX                      | Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.     |
| VEX.NDS.128.66.0F C4 /r ib<br>VPINSRW xmm1, xmm2, r32/m16, imm8       | RVMI    | V/V                          | AVX                      | Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.   |
| VEX.NDS.128.66.0F3A.W0 22 /r ib<br>VPINSRD xmm1, xmm2, r32/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<br>VPINSRQ xmm1, xmm2, r64/m64, imm8  | RVMI    | V/N.E. <sup>1</sup>          | AVX                      | Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8. |
| EVEX.NDS.128.66.0F3A.WIG 20 /r ib<br>VPINSRB xmm1, xmm2, r32/m8, imm8 | T1S     | V/V                          | AVX512BW                 | Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.     |
| EVEX.NDS.128.66.0F.WIG C4 /r ib<br>VPINSRW xmm1, xmm2, r32/m16, imm8  | T1S     | V/V                          | AVX512BW                 | Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.   |
| EVEX.NDS.128.66.0F3A.W0 22 /r ib<br>VPINSRD xmm1, xmm2, r32/m32, imm8 | T1S     | V/V                          | AVX512DQ                 | Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8. |
| EVEX.NDS.128.66.0F3A.W1 22 /r ib<br>VPINSRQ xmm1, xmm2, r64/m64, imm8 | T1S     | V/N.E. <sup>1</sup>          | AVX512DQ                 | Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8. |

**NOTES:**

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

**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      |
| T1S   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

## Description

Copies a byte/word/dword/qword from the second source operand and inserts it into the destination operand at the byte/word/dword/qword offset specified with the immediate operand (third operand). The other bytes/words/dwords/qwords in the destination register are copied from the first source operand. The byte select is specified by the 4/3/2/1 least-significant bits of the immediate.

The first source operand and destination operands are XMM registers. The second source operand is a r32 register or an 8-/16-/32-/ or 64-bit memory location. For PINSRW, REX.W causes the source to be an r64 instead of an r32. REX.W distinguishes between PINSRD and PINSRQ (PINSRQ is not encodable in 32-bit modes).

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

## Operation

**write\_q\_element(position, val, src)**

```
{
TEMP ← SRC
CASE (position)
0: TEMP[63:0] ← val
1: TEMP[127:64] ← val
ESAC
return TEMP
}
```

**write\_d\_element(position, val, src)**

```
{
TEMP ← SRC
CASE (position)
0: TEMP[31:0] ← val
1: TEMP[63:32] ← val
2: TEMP[95:64] ← val
3: TEMP[127:96] ← val
ESAC
return TEMP
}
```

**write\_w\_element(position, val, src)**

```
{
TEMP ← SRC
CASE (position)
0: TEMP[15:0] ← val
1: TEMP[31:16] ← val
2: TEMP[47:32] ← val
3: TEMP[63:48] ← val
4: TEMP[79:64] ← val
5: TEMP[95:80] ← val
6: TEMP[111:96] ← val
7: TEMP[127:112] ← val
ESAC
return TEMP
}
```

**write\_b\_element(position, val, src)**

```

{
TEMP ← SRC
CASE (position)
0: TEMP[7:0] ← val
1: TEMP[15:8] ← val
2: TEMP[23:16] ← val
3: TEMP[31:24] ← val
4: TEMP[39:32] ← val
5: TEMP[47:40] ← val
6: TEMP[55:48] ← val
7: TEMP[63:56] ← val
8: TEMP[71:64] ← val
9: TEMP[79:72] ← val
10: TEMP[87:80] ← val
11: TEMP[95:88] ← val
12: TEMP[103:96] ← val
13: TEMP[111:104] ← val
14: TEMP[119:112] ← val
15: TEMP[127:120] ← val
ESAC
return TEMP
}

```

**VPINSRQ (EVEX encoded version)**

```

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[MAX_VL-1:128] ← 0

```

**VPINSRD (EVEX encoded version)**

```

SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[MAX_VL-1:128] ← 0

```

**VPINSRW (EVEX encoded version)**

```

SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[MAX_VL-1:128] ← 0

```

**VPINSRB (EVEX encoded version)**

```

SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[MAX_VL-1:128] ← 0

```

**VPINSRQ (VEX.128 encoded version)**

```

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[MAX_VL-1:128] ← 0

```

**VPINSRD (VEX.128 encoded version)**

```

SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[MAX_VL-1:128] ← 0

```

**VPINSRW (VEX.128 encoded version)**

SEL  $\leftarrow$  imm8[2:0]  
 DEST[127:0]  $\leftarrow$  write\_w\_element(SEL, SRC2, SRC1)  
 DEST[MAX\_VL-1:128]  $\leftarrow$  0

**VPINSRB (VEX.128 encoded version)**

SEL  $\leftarrow$  imm8[3:0]  
 DEST[127:0]  $\leftarrow$  write\_b\_element(SEL, SRC2, SRC1)  
 DEST[MAX\_VL-1:128]  $\leftarrow$  0

**PINSRQ (Legacy SSE version)**

SEL  $\leftarrow$  imm8[0]  
 DEST[127:0]  $\leftarrow$  write\_q\_element(SEL, SRC, DEST)  
 DEST[MAX\_VL-1:128] (Unmodified)

**PINSRD (Legacy SSE version)**

SEL  $\leftarrow$  imm8[1:0]  
 DEST[127:0]  $\leftarrow$  write\_d\_element(SEL, SRC, DEST)  
 DEST[MAX\_VL-1:128] (Unmodified)

**PINSRW (Legacy SSE version)**

SEL  $\leftarrow$  imm8[2:0]  
 DEST[127:0]  $\leftarrow$  write\_w\_element(SEL, SRC, DEST)  
 DEST[MAX\_VL-1:128] (Unmodified)

**PINSRB (Legacy SSE version)**

SEL  $\leftarrow$  imm8[3:0]  
 DEST[127:0]  $\leftarrow$  write\_b\_element(SEL, SRC, DEST)  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

PINSRB `__m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);`  
 PINSRW `__m128i _mm_insert_epi16 (__m128i a, int b, int imm)`  
 PINSRD `__m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);`  
 PINSRQ `__m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type 5;  
 EVEX-encoded instruction, see Exceptions Type E9NF.  
 #UD If VEX.L = 1 or EVEX.L'L > 0.

## VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators

| Opcode/<br>Instruction  | Op/En | 32/64<br>bit Mode<br>Support | CPUID                  | Description   |
|---|-------|------------------------------|------------------------|---|
| EVEX.DDS.128.66.0F38.W1 B4 /r<br>VPMADD52LUQ xmm1 {k1}{z},<br>xmm2,xmm3/m128/m64bcst  | FV    | V/V                          | AVX512IFMA<br>AVX512VL | Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W1 B4 /r<br>VPMADD52LUQ ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | FV    | V/V                          | AVX512IFMA<br>AVX512VL | Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W1 B4 /r<br>VPMADD52LUQ zmm1 {k1}{z},<br>zmm2,zmm3/m512/m64bcst  | FV    | V/V                          | AVX512IFMA             | Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3    | Operand 4 |
|-------|------------------|---------------|--------------|-----------|
| FV    | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM.r/m(r) | NA        |

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

**Operation****VPMADD52LUQ (EVEX encoded)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64;

IF k1[j] OR \*no writemask\* THEN

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] ← ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] ← ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] ← ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] ← DEST[i+63:i] + ZeroExtend64(temp128[51:0]);

DEST[i+63:i] ← Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] ← 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADD52LUQ \_\_m512i \_\_mm512\_madd52lo\_epu64(\_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_mask\_madd52lo\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_maskz\_madd52lo\_epu64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_madd52lo\_epu64(\_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_mask\_madd52lo\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_maskz\_madd52lo\_epu64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m128i \_\_mm\_madd52lo\_epu64(\_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_mask\_madd52lo\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_maskz\_madd52lo\_epu64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit Accumulators'

| Opcode/<br>Instruction  | Op/En | 32/64<br>bit Mode<br>Support | CPUID                  | Description  |
|---|-------|------------------------------|------------------------|--|
| EVEX.DDS.128.66.0F38.W1 B5 /r<br>VPMADD52HUQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV    | V/V                          | AVX512IFMA<br>AVX512VL | Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1. |
| EVEX.DDS.256.66.0F38.W1 B5 /r<br>VPMADD52HUQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV    | V/V                          | AVX512IFMA<br>AVX512VL | Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1. |
| EVEX.DDS.512.66.0F38.W1 B5 /r<br>VPMADD52HUQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV    | V/V                          | AVX512IFMA             | Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3    | Operand 4 |
|-------|------------------|---------------|--------------|-----------|
| FV    | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM.r/m(r) | NA        |

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.



**Operation****VPMADD52HUQ (EVEX encoded)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64;

IF k1[j] OR \*no writemask\* THEN

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] ← ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] ← ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] ← ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] ← DEST[i+63:i] + ZeroExtend64(temp128[103:52]);

DEST[i+63:i] ← Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] ← 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADD52HUQ \_\_m512i \_\_mm512\_madd52hi\_epu64( \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_mask\_madd52hi\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_maskz\_madd52hi\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_madd52hi\_epu64( \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_mask\_madd52hi\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_maskz\_madd52hi\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m128i \_\_mm\_madd52hi\_epu64( \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_mask\_madd52hi\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_maskz\_madd52hi\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 38 3C /r<br>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.                    |
| 66 0F EE /r<br>PMAXSW xmm1, xmm2/m128                                     | RM         | V/V                          | SSE2                     | Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.                   |
| 66 0F 38 3D /r<br>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 3C /r<br>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.128.66.0F.WIG EE /r<br>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.128.66.0F38.WIG 3D /r<br>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 3C /r<br>VPMAXSB ymm1, ymm2, ymm3/m256            | RVM        | V/V                          | AVX2                     | Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.                    |
| VEX.NDS.256.66.0F.WIG EE /r<br>VPMAXSW ymm1, ymm2, ymm3/m256              | RVM        | V/V                          | AVX2                     | Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.                    |
| VEX.NDS.256.66.0F38.WIG 3D /r<br>VPMAXSD ymm1, ymm2, ymm3/m256            | RVM        | V/V                          | AVX2                     | Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.                   |
| EVEX.NDS.128.66.0F38.WIG 3C /r<br>VPMAXSB xmm1{k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.WIG 3C /r<br>VPMAXSB ymm1{k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.WIG 3C /r<br>VPMAXSB zmm1{k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F.WIG EE /r<br>VPMAXSW xmm1{k1}{z}, xmm2,<br>xmm3/m128   | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG EE /r<br>VPMAXSW ymm1{k1}{z}, ymm2,<br>ymm3/m256   | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1. |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.NDS.512.66.0F.WIG EE /r<br>VPMAXSW zmm1{k1}{z}, zmm2,<br>zmm3/m512           | FVM        | V/V                          | AVX512BW                 | Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.          |
| EVEX.NDS.128.66.0F38.W0 3D /r<br>VPMAXSD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F38.W0 3D /r<br>VPMAXSD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 3D /r<br>VPMAXSD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1. |
| EVEX.NDS.128.66.0F38.W1 3D /r<br>VPMAXSQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F38.W1 3D /r<br>VPMAXSQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F38.W1 3D /r<br>VPMAXSQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

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 (MAX\_VL-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 (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

### Operation

#### PMAXSB (128-bit Legacy SSE version)

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

#### VPMAXSB (VEX.128 encoded version)

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

#### VPMAXSB (VEX.256 encoded version)

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] > SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMAXSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+7:i] &gt; SRC2[i+7:i]

THEN DEST[i+7:i] ← SRC1[i+7:i];

ELSE DEST[i+7:i] ← SRC2[i+7:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**PMAxSw (128-bit Legacy SSE version)**

IF DEST[15:0] &gt; 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] &gt; SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

**VPMAXSw (VEX.128 encoded version)**

IF SRC1[15:0] &gt; 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] &gt; SRC2[127:112] THEN

DEST[127:112] ← SRC1[127:112];

ELSE

DEST[127:112] ← SRC2[127:112]; FI;

DEST[MAX\_VL-1:128] ← 0

**VPMAXSw (VEX.256 encoded version)**

IF SRC1[15:0] &gt; 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] &gt; SRC2[255:240] THEN

DEST[255:240] ← SRC1[255:240];

ELSE

DEST[255:240] ← SRC2[255:240]; FI;

DEST[MAX\_VL-1:256] ← 0

**VPMAXSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+15:i] &gt; SRC2[i+15:i]

THEN DEST[i+15:i] ← SRC1[i+15:i];

ELSE DEST[i+15:i] ← SRC2[i+15:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**PMAXSD (128-bit Legacy SSE version)**

IF DEST[31:0] &gt; SRC[31:0] THEN

DEST[31:0] ← DEST[31:0];

ELSE

DEST[31:0] ← SRC[31:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF DEST[127:96] &gt; SRC[127:96] THEN

DEST[127:96] ← DEST[127:96];

ELSE

DEST[127:96] ← SRC[127:96]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

**VPMAXSD (VEX.128 encoded version)**

IF SRC1[31:0] &gt; SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 3rd dwords in source and destination operands \*)

IF SRC1[127:96] &gt; SRC2[127:96] THEN

DEST[127:96] ← SRC1[127:96];

ELSE

DEST[127:96] ← SRC2[127:96]; FI;

DEST[MAX\_VL-1:128] ← 0

**VPMAXSD (VEX.256 encoded version)**

IF SRC1[31:0] &gt; 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] &gt; SRC2[255:224] THEN

DEST[255:224] ← SRC1[255:224];

ELSE

DEST[255:224] ← SRC2[255:224]; FI;

DEST[MAX\_VL-1:256] ← 0

**VPMAXSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+31:i] > SRC2[31:0]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[31:0];
        FI;
      ELSE
        IF SRC1[j+31:i] > SRC2[i+31:i]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[i+31:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
        ELSE DEST[j+31:i] ← 0 ; zeroing-masking
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VPMAXSQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+63:i] > SRC2[63:0]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[63:0];
        FI;
      ELSE
        IF SRC1[j+63:i] > SRC2[i+63:i]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[i+63:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
        ELSE ; zeroing-masking
          THEN DEST[j+63:i] ← 0
        FI
    FI;
  ENDFOR;
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMSXSB \_\_m512i \_\_mm512\_max\_epi8( \_\_m512i a, \_\_m512i b);  
 VPMSXSB \_\_m512i \_\_mm512\_mask\_max\_epi8( \_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSB \_\_m512i \_\_mm512\_maskz\_max\_epi8( \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSW \_\_m512i \_\_mm512\_max\_epi16( \_\_m512i a, \_\_m512i b);  
 VPMSXSW \_\_m512i \_\_mm512\_mask\_max\_epi16( \_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSW \_\_m512i \_\_mm512\_maskz\_max\_epi16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSB \_\_m256i \_\_mm256\_mask\_max\_epi8( \_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSB \_\_m256i \_\_mm256\_maskz\_max\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSW \_\_m256i \_\_mm256\_mask\_max\_epi16( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSW \_\_m256i \_\_mm256\_maskz\_max\_epi16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSB \_\_m128i \_\_mm\_mask\_max\_epi8( \_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSB \_\_m128i \_\_mm\_maskz\_max\_epi8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSW \_\_m128i \_\_mm\_mask\_max\_epi16( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSW \_\_m128i \_\_mm\_maskz\_max\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSD \_\_m256i \_\_mm256\_mask\_max\_epi32( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSD \_\_m256i \_\_mm256\_maskz\_max\_epi32( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSQ \_\_m256i \_\_mm256\_mask\_max\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSQ \_\_m256i \_\_mm256\_maskz\_max\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMSXSD \_\_m128i \_\_mm\_mask\_max\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSD \_\_m128i \_\_mm\_maskz\_max\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSQ \_\_m128i \_\_mm\_mask\_max\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSQ \_\_m128i \_\_mm\_maskz\_max\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMSXSD \_\_m512i \_\_mm512\_max\_epi32( \_\_m512i a, \_\_m512i b);  
 VPMSXSD \_\_m512i \_\_mm512\_mask\_max\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSD \_\_m512i \_\_mm512\_maskz\_max\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSQ \_\_m512i \_\_mm512\_max\_epi64( \_\_m512i a, \_\_m512i b);  
 VPMSXSQ \_\_m512i \_\_mm512\_mask\_max\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMSXSQ \_\_m512i \_\_mm512\_maskz\_max\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 (V)VPMSXSB \_\_m128i \_\_mm\_max\_epi8( \_\_m128i a, \_\_m128i b);  
 (V)VPMSXSW \_\_m128i \_\_mm\_max\_epi16( \_\_m128i a, \_\_m128i b);  
 (V)VPMSXSD \_\_m128i \_\_mm\_max\_epi32( \_\_m128i a, \_\_m128i b);  
 VPMSXSB \_\_m256i \_\_mm256\_max\_epi8( \_\_m256i a, \_\_m256i b);  
 VPMSXSW \_\_m256i \_\_mm256\_max\_epi16( \_\_m256i a, \_\_m256i b);  
 VPMSXSD \_\_m256i \_\_mm256\_max\_epi32( \_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded VPMSXSD/Q, see Exceptions Type E4.  
 EVEX-encoded VPMSXSB/W, see Exceptions Type E4.nb.



## PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F DE /r<br>PMAXUB xmm1, xmm2/m128                                     | RM         | V/V                          | SSE2                     | Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.                    |
| 66 0F 38 3E/r<br>PMAXUW xmm1, xmm2/m128                                   | RM         | V/V                          | SSE4_1                   | Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.                   |
| VEX.NDS.128.66.0F DE /r<br>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.128.66.0F38 3E/r<br>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.256.66.0F DE /r<br>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.                    |
| VEX.NDS.256.66.0F38 3E/r<br>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.                    |
| EVEX.NDS.128.66.0F.WIG DE /r<br>VPMAXUB xmm1{k1}{z}, xmm2,<br>xmm3/m128   | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG DE /r<br>VPMAXUB ymm1{k1}{z}, ymm2,<br>ymm3/m256   | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.WIG DE /r<br>VPMAXUB zmm1{k1}{z}, zmm2,<br>zmm3/m512   | FVM        | V/V                          | AVX512BW                 | Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F38.WIG 3E /r<br>VPMAXUW xmm1{k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.WIG 3E /r<br>VPMAXUW ymm1{k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.WIG 3E /r<br>VPMAXUW zmm1{k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMAXUB (128-bit Legacy SSE version)**

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMAXUB (VEX.128 encoded version)**

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMAXUB (VEX.256 encoded version)**

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] > SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMAXUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+7:i] &gt; SRC2[i+7:i]

THEN DEST[i+7:i] ← SRC1[i+7:i];

ELSE DEST[i+7:i] ← SRC2[i+7:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**PMAUW (128-bit Legacy SSE version)**

IF DEST[15:0] &gt; 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] &gt; SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

**VPMAXUW (VEX.128 encoded version)**

IF SRC1[15:0] &gt; 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] &gt; SRC2[127:112] THEN

DEST[127:112] ← SRC1[127:112];

ELSE

DEST[127:112] ← SRC2[127:112]; FI;

DEST[MAX\_VL-1:128] ← 0

**VPMAXUW (VEX.256 encoded version)**

IF SRC1[15:0] &gt; 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] &gt; SRC2[255:240] THEN

DEST[255:240] ← SRC1[255:240];

ELSE

DEST[255:240] ← SRC2[255:240]; FI;

DEST[MAX\_VL-1:128] ← 0

**VPMAXUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+15:i] &gt; SRC2[i+15:i]

THEN DEST[i+15:i] ← SRC1[i+15:i];

ELSE DEST[i+15:i] ← SRC2[i+15:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMAXUB \_\_m512i\_mm512\_max\_epu8(\_\_m512i a, \_\_m512i b);

VPMAXUB \_\_m512i\_mm512\_mask\_max\_epu8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMAXUB \_\_m512i\_mm512\_maskz\_max\_epu8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMAXUW \_\_m512i\_mm512\_max\_epu16(\_\_m512i a, \_\_m512i b);

VPMAXUW \_\_m512i\_mm512\_mask\_max\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMAXUW \_\_m512i\_mm512\_maskz\_max\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMAXUB \_\_m256i\_mm256\_mask\_max\_epu8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMAXUB \_\_m256i\_mm256\_maskz\_max\_epu8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMAXUW \_\_m256i\_mm256\_mask\_max\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMAXUW \_\_m256i\_mm256\_maskz\_max\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMAXUB \_\_m128i\_mm\_mask\_max\_epu8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMAXUB \_\_m128i\_mm\_maskz\_max\_epu8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMAXUW \_\_m128i\_mm\_mask\_max\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMAXUW \_\_m128i\_mm\_maskz\_max\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMAXUB \_\_m128i\_mm\_max\_epu8 (\_\_m128i a, \_\_m128i b);

(V)PMAXUW \_\_m128i\_mm\_max\_epu16 (\_\_m128i a, \_\_m128i b)

VPMAXUB \_\_m256i\_mm256\_max\_epu8 (\_\_m256i a, \_\_m256i b);

VPMAXUW \_\_m256i\_mm256\_max\_epu16 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| 66 0F 38 3F /r<br>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<br>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<br>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.                            |
| EVEX.NDS.128.66.0F38.W0 3F /r<br>VPMAXUD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.W0 3F /r<br>VPMAXUD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.W0 3F /r<br>VPMAXUD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                         | AVX512F                  | Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F38.W1 3F /r<br>VPMAXUQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.W1 3F /r<br>VPMAXUQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.W1 3F /r<br>VPMAXUQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                         | AVX512F                  | Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMAXUD (128-bit Legacy SSE version)

```
IF DEST[31:0] > SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] > SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

### VPMAXUD (VEX.128 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] > SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAX_VL-1:128] ← 0
```

### VPMAXUD (VEX.256 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMAXUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+31:i] > SRC2[31:0]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[31:0];
        FI;
      ELSE
        IF SRC1[j+31:i] > SRC2[i+31:i]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[i+31:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
      ELSE ; zeroing-masking
        THEN DEST[j+31:i] ← 0
      FI
    FI;
  ENDFOR;
  DEST[MAX_VL-1:VL] ← 0

```

**VPMAXUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+63:i] > SRC2[63:0]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[63:0];
        FI;
      ELSE
        IF SRC1[j+31:i] > SRC2[i+31:i]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[i+63:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
      ELSE ; zeroing-masking
        THEN DEST[j+63:i] ← 0
      FI
    FI;
  ENDFOR;
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMAXUD \_\_m512i \_\_mm512\_max\_epu32(\_\_m512i a, \_\_m512i b);  
 VPMAXUD \_\_m512i \_\_mm512\_mask\_max\_epu32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUD \_\_m512i \_\_mm512\_maskz\_max\_epu32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUQ \_\_m512i \_\_mm512\_max\_epu64(\_\_m512i a, \_\_m512i b);  
 VPMAXUQ \_\_m512i \_\_mm512\_mask\_max\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUQ \_\_m512i \_\_mm512\_maskz\_max\_epu64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUD \_\_m256i \_\_mm256\_mask\_max\_epu32(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUD \_\_m256i \_\_mm256\_maskz\_max\_epu32(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUQ \_\_m256i \_\_mm256\_mask\_max\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUQ \_\_m256i \_\_mm256\_maskz\_max\_epu64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUD \_\_m128i \_\_mm\_mask\_max\_epu32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMAXUD \_\_m128i \_\_mm\_maskz\_max\_epu32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMAXUQ \_\_m128i \_\_mm\_mask\_max\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMAXUQ \_\_m128i \_\_mm\_maskz\_max\_epu64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMAXUD \_\_m128i \_\_mm\_max\_epu32 (\_\_m128i a, \_\_m128i b);  
 VPMAXUD \_\_m256i \_\_mm256\_max\_epu32 (\_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## PMINSB/PMINSW—Minimum of Packed Signed Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| 66 0F 38 38 /r<br>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.                    |
| 66 0F EA /r<br>PMINSW xmm1, xmm2/m128                                     | RM        | V/V                          | SSE2                     | Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.                    |
| VEX.NDS.128.66.0F38 38 /r<br>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.128.66.0F EA /r<br>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.256.66.0F38 38 /r<br>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.                    |
| VEX.NDS.256.66.0F EA /r<br>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.                   |
| EVEX.NDS.128.66.0F38.WIG 38 /r<br>VPMINSB xmm1{k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.WIG 38 /r<br>VPMINSB ymm1{k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.WIG 38 /r<br>VPMINSB zmm1{k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F.WIG EA /r<br>VPMINSW xmm1{k1}{z}, xmm2,<br>xmm3/m128   | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG EA /r<br>VPMINSW ymm1{k1}{z}, ymm2,<br>ymm3/m256   | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.WIG EA /r<br>VPMINSW zmm1{k1}{z}, zmm2,<br>zmm3/m512   | FVM       | V/V                          | AVX512BW                 | Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMINSB (128-bit Legacy SSE version)**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMINSB (VEX.128 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMINSB (VEX.256 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] < SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMINSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\* THEN

IF SRC1[j+7:i] &lt; SRC2[j+7:i]

THEN DEST[j+7:i] ← SRC1[j+7:i];

ELSE DEST[j+7:i] ← SRC2[j+7:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+7:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+7:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**PMINSW (128-bit Legacy SSE version)**

IF DEST[15:0] &lt; 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] &lt; SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

**VPMINSW (VEX.128 encoded version)**

IF SRC1[15:0] &lt; 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] &lt; SRC2[127:112] THEN

DEST[127:112] ← SRC1[127:112];

ELSE

DEST[127:112] ← SRC2[127:112]; FI;

DEST[MAX\_VL-1:128] ← 0

**VPMINSW (VEX.256 encoded version)**

IF SRC1[15:0] &lt; 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] &lt; SRC2[255:240] THEN

DEST[255:240] ← SRC1[255:240];

ELSE

DEST[255:240] ← SRC2[255:240]; FI;

DEST[MAX\_VL-1:256] ← 0

**VPMINSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+15:i] &lt; SRC2[i+15:i]

THEN DEST[i+15:i] ← SRC1[i+15:i];

ELSE DEST[i+15:i] ← SRC2[i+15:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINSB \_\_m512i\_mm512\_min\_epi8( \_\_m512i a, \_\_m512i b);

VPMINSB \_\_m512i\_mm512\_mask\_min\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMINSB \_\_m512i\_mm512\_maskz\_min\_epi8( \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMINSW \_\_m512i\_mm512\_min\_epi16( \_\_m512i a, \_\_m512i b);

VPMINSW \_\_m512i\_mm512\_mask\_min\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMINSW \_\_m512i\_mm512\_maskz\_min\_epi16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMINSB \_\_m256i\_mm256\_mask\_min\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMINSB \_\_m256i\_mm256\_maskz\_min\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMINSW \_\_m256i\_mm256\_mask\_min\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMINSW \_\_m256i\_mm256\_maskz\_min\_epi16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMINSB \_\_m128i\_mm\_mask\_min\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMINSB \_\_m128i\_mm\_maskz\_min\_epi8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMINSW \_\_m128i\_mm\_mask\_min\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMINSW \_\_m128i\_mm\_maskz\_min\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMINSB \_\_m128i\_mm\_min\_epi8 ( \_\_m128i a, \_\_m128i b);

(V)PMINSW \_\_m128i\_mm\_min\_epi16 ( \_\_m128i a, \_\_m128i b)

VPMINSB \_\_m256i\_mm256\_min\_epi8 ( \_\_m256i a, \_\_m256i b);

VPMINSW \_\_m256i\_mm256\_min\_epi16 ( \_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMINSD/PMINSQ—Minimum of Packed Signed Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| 66 0F 38 39 /r<br>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<br>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<br>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.                            |
| EVEX.NDS.128.66.0F38.W0 39 /r<br>VPMINSD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.         |
| EVEX.NDS.256.66.0F38.W0 39 /r<br>VPMINSD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.         |
| EVEX.NDS.512.66.0F38.W0 39 /r<br>VPMINSD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F38.W1 39 /r<br>VPMINSQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.         |
| EVEX.NDS.256.66.0F38.W1 39 /r<br>VPMINSQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.         |
| EVEX.NDS.512.66.0F38.W1 39 /r<br>VPMINSQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMINSD (128-bit Legacy SSE version)

```
IF DEST[31:0] < SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] < SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

### VPMINSD (VEX.128 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] < SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAX_VL-1:128] ← 0
```

### VPMINSD (VEX.256 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMINSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+31:i] &lt; SRC2[31:0]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[j+31:i] &lt; SRC2[i+31:i]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPMINSQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+63:i] &lt; SRC2[63:0]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[j+63:i] &lt; SRC2[i+63:i]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINSD \_\_m512i \_mm512\_min\_epi32( \_\_m512i a, \_\_m512i b);  
 VPMINSD \_\_m512i \_mm512\_mask\_min\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINSD \_\_m512i \_mm512\_maskz\_min\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINSQ \_\_m512i \_mm512\_min\_epi64( \_\_m512i a, \_\_m512i b);  
 VPMINSQ \_\_m512i \_mm512\_mask\_min\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINSQ \_\_m512i \_mm512\_maskz\_min\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINSD \_\_m256i \_mm256\_mask\_min\_epi32( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINSD \_\_m256i \_mm256\_maskz\_min\_epi32( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINSQ \_\_m256i \_mm256\_mask\_min\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINSQ \_\_m256i \_mm256\_maskz\_min\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINSD \_\_m128i \_mm\_mask\_min\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINSD \_\_m128i \_mm\_maskz\_min\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINSQ \_\_m128i \_mm\_mask\_min\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINSQ \_\_m128i \_mm\_maskz\_min\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMINSD \_\_m128i \_mm\_min\_epi32( \_\_m128i a, \_\_m128i b);  
 VPMINSD \_\_m256i \_mm256\_min\_epi32( \_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## PMINUB/PMINUW—Minimum of Packed Unsigned Integers

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F DA /r<br>PMINUB xmm1, xmm2/m128                                | RM         | V/V                          | SSE2                     | Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.                     |
| 66 0F 38 3A/r<br>PMINUW xmm1, xmm2/m128                              | RM         | V/V                          | SSE4_1                   | Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.                     |
| VEX.NDS.128.66.0F DA /r<br>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.128.66.0F38 3A/r<br>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.256.66.0F DA /r<br>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.                     |
| VEX.NDS.256.66.0F38 3A/r<br>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.                    |
| EVEX.NDS.128.66.0F DA /r<br>VPMINUB xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.  |
| EVEX.NDS.256.66.0F DA /r<br>VPMINUB ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.  |
| EVEX.NDS.512.66.0F DA /r<br>VPMINUB zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.  |
| EVEX.NDS.128.66.0F38 3A/r<br>VPMINUW xmm1{k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38 3A/r<br>VPMINUW ymm1{k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38 3A/r<br>VPMINUW zmm1{k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMINUB (128-bit Legacy SSE version)****PMINUB instruction for 128-bit operands:**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**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[MAX_VL-1:128] ← 0
```

**VPMINUB (VEX.256 encoded version)****VPMINUB instruction for 256-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;
```

DEST[MAX\_VL-1:256] ← 0

#### VPMINUB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\* THEN

    IF SRC1[j+7:i] < SRC2[j+7:i]

      THEN DEST[j+7:i] ← SRC1[j+7:i];

      ELSE DEST[j+7:i] ← SRC2[j+7:i];

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[j+7:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[j+7:i] ← 0

  FI

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

#### PMINUW (128-bit Legacy SSE version)

##### PMINUW instruction for 128-bit operands:

  IF DEST[15:0] < SRC[15:0] THEN

    DEST[15:0] ← DEST[15:0];

  ELSE

    DEST[15:0] ← SRC[15:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

  IF DEST[127:112] < SRC[127:112] THEN

    DEST[127:112] ← DEST[127:112];

  ELSE

    DEST[127:112] ← SRC[127:112]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

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[MAX\_VL-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;
DEST[MAX_VL-1:256] ← 0

```

**VPMINUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+15:i] &lt; SRC2[i+15:i]

THEN DEST[i+15:i] ← SRC1[i+15:i];

ELSE DEST[i+15:i] ← SRC2[i+15:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINUB \_\_m512i\_mm512\_min\_epu8( \_\_m512i a, \_\_m512i b);

VPMINUB \_\_m512i\_mm512\_mask\_min\_epu8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMINUB \_\_m512i\_mm512\_maskz\_min\_epu8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMINUW \_\_m512i\_mm512\_min\_epu16( \_\_m512i a, \_\_m512i b);

VPMINUW \_\_m512i\_mm512\_mask\_min\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMINUW \_\_m512i\_mm512\_maskz\_min\_epu16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMINUB \_\_m256i\_mm256\_mask\_min\_epu8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMINUB \_\_m256i\_mm256\_maskz\_min\_epu8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMINUW \_\_m256i\_mm256\_mask\_min\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMINUW \_\_m256i\_mm256\_maskz\_min\_epu16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMINUB \_\_m128i\_mm\_mask\_min\_epu8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMINUB \_\_m128i\_mm\_maskz\_min\_epu8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMINUW \_\_m128i\_mm\_mask\_min\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMINUW \_\_m128i\_mm\_maskz\_min\_epu16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMINUB \_\_m128i\_mm\_min\_epu8 ( \_\_m128i a, \_\_m128i b)

(V)PMINUW \_\_m128i\_mm\_min\_epu16 ( \_\_m128i a, \_\_m128i b);

VPMINUB \_\_m256i\_mm256\_min\_epu8 ( \_\_m256i a, \_\_m256i b)

VPMINUW \_\_m256i\_mm256\_min\_epu16 ( \_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| 66 0F 38 3B /r<br>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<br>VPMINUD xmm1, xmm2,<br>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<br>VPMINUD ymm1, ymm2,<br>ymm3/m256                 | RVM       | V/V                          | AVX2                     | Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.                            |
| EVEX.NDS.128.66.0F38.W0 3B /r<br>VPMINUD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.W0 3B /r<br>VPMINUD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.W0 3B /r<br>VPMINUD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F38.W1 3B /r<br>VPMINUQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.W1 3B /r<br>VPMINUQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.W1 3B /r<br>VPMINUQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMINUD (128-bit Legacy SSE version)

PMINUD instruction for 128-bit operands:

IF DEST[31:0] < SRC[31:0] THEN

DEST[31:0] ← DEST[31:0];

ELSE

DEST[31:0] ← SRC[31:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF DEST[127:96] < SRC[127:96] THEN

DEST[127:96] ← DEST[127:96];

ELSE

DEST[127:96] ← SRC[127:96]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

### VPMINUD (VEX.128 encoded version)

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 3rd dwords in source and destination operands \*)

IF SRC1[127:96] < SRC2[127:96] THEN

DEST[127:96] ← SRC1[127:96];

ELSE

DEST[127:96] ← SRC2[127:96]; FI;

DEST[MAX\_VL-1:128] ← 0

### VPMINUD (VEX.256 encoded version)

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 7th dwords in source and destination operands \*)

IF SRC1[255:224] < SRC2[255:224] THEN

DEST[255:224] ← SRC1[255:224];

ELSE

DEST[255:224] ← SRC2[255:224]; FI;

DEST[MAX\_VL-1:256] ← 0

**VPMINUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+31:i] &lt; SRC2[31:0]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[j+31:i] &lt; SRC2[i+31:i]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPMINUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+63:i] &lt; SRC2[63:0]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[j+63:i] &lt; SRC2[i+63:i]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINUD \_\_m512i \_mm512\_min\_epu32( \_\_m512i a, \_\_m512i b);  
 VPMINUD \_\_m512i \_mm512\_mask\_min\_epu32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINUD \_\_m512i \_mm512\_maskz\_min\_epu32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINUQ \_\_m512i \_mm512\_min\_epu64( \_\_m512i a, \_\_m512i b);  
 VPMINUQ \_\_m512i \_mm512\_mask\_min\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINUQ \_\_m512i \_mm512\_maskz\_min\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINUD \_\_m256i \_mm256\_mask\_min\_epu32(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINUD \_\_m256i \_mm256\_maskz\_min\_epu32( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINUQ \_\_m256i \_mm256\_mask\_min\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINUQ \_\_m256i \_mm256\_maskz\_min\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINUD \_\_m128i \_mm\_mask\_min\_epu32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINUD \_\_m128i \_mm\_maskz\_min\_epu32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINUQ \_\_m128i \_mm\_mask\_min\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINUQ \_\_m128i \_mm\_maskz\_min\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMINUD \_\_m128i \_mm\_min\_epu32 ( \_\_m128i a, \_\_m128i b);  
 VPMINUD \_\_m256i \_mm256\_min\_epu32 ( \_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

| Opcode/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 28 /r<br>VPMOVM2B xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.256.F3.0F38.W0 28 /r<br>VPMOVM2B ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.512.F3.0F38.W0 28 /r<br>VPMOVM2B zmm1, k1 | RM        | V/V                          | AVX512BW                 | Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.128.F3.0F38.W1 28 /r<br>VPMOVM2W xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.256.F3.0F38.W1 28 /r<br>VPMOVM2W ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.512.F3.0F38.W1 28 /r<br>VPMOVM2W zmm1, k1 | RM        | V/V                          | AVX512BW                 | Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.128.F3.0F38.W0 38 /r<br>VPMOVM2D xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1. |
| EVEX.256.F3.0F38.W0 38 /r<br>VPMOVM2D ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1. |
| EVEX.512.F3.0F38.W0 38 /r<br>VPMOVM2D zmm1, k1 | RM        | V/V                          | AVX512DQ                 | Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1. |
| EVEX.128.F3.0F38.W1 38 /r<br>VPMOVM2Q xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.   |
| EVEX.256.F3.0F38.W1 38 /r<br>VPMOVM2Q ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.   |
| EVEX.512.F3.0F38.W1 38 /r<br>VPMOVM2Q zmm1, k1 | RM        | V/V                          | AVX512DQ                 | Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVM2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF SRC[j]

THEN DEST[i+7:i] ← -1

ELSE DEST[i+7:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVM2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF SRC[j]

THEN DEST[i+15:i] ← -1

ELSE DEST[i+15:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVM2D (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF SRC[j]

THEN DEST[i+31:i] ← -1

ELSE DEST[i+31:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVM2Q (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF SRC[j]

THEN DEST[i+63:i] ← -1

ELSE DEST[i+63:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVM2B __m512i __mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i __mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i __mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i __mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i __mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i __mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i __mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i __mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i __mm_movm_epi8(__mmask16);
VPMOVM2D __m128i __mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i __mm_movm_epi64(__mmask8);
VPMOVM2W __m128i __mm_movm_epi16(__mmask8);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                    If EVEX.vvvv != 1111B.

**VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask**

| Opcode/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 29 /r<br>VPMOVB2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.       |
| EVEX.256.F3.0F38.W0 29 /r<br>VPMOVB2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.       |
| EVEX.512.F3.0F38.W0 29 /r<br>VPMOVB2M k1, zmm1 | RM        | V/V                          | AVX512BW                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.       |
| EVEX.128.F3.0F38.W1 29 /r<br>VPMOVW2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.       |
| EVEX.256.F3.0F38.W1 29 /r<br>VPMOVW2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.       |
| EVEX.512.F3.0F38.W1 29 /r<br>VPMOVW2M k1, zmm1 | RM        | V/V                          | AVX512BW                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.       |
| EVEX.128.F3.0F38.W0 39 /r<br>VPMOVD2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1. |
| EVEX.256.F3.0F38.W0 39 /r<br>VPMOVD2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1. |
| EVEX.512.F3.0F38.W0 39 /r<br>VPMOVD2M k1, zmm1 | RM        | V/V                          | AVX512DQ                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1. |
| EVEX.128.F3.0F38.W1 39 /r<br>VPMOVQ2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.   |
| EVEX.256.F3.0F38.W1 39 /r<br>VPMOVQ2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.   |
| EVEX.512.F3.0F38.W1 39 /r<br>VPMOVQ2M k1, zmm1 | RM        | V/V                          | AVX512DQ                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.   |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

**Description**

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVB2M (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 8
  IF SRC[i+7]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VPMOVW2M (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF SRC[i+15]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VPMOVD2M (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF SRC[i+31]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VPMOVQ2M (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF SRC[i+63]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMPPOVB2M __mmask64 _mm512_movepi8_mask( __m512i );
VPMPPOVD2M __mmask16 _mm512_movepi32_mask( __m512i );
VPMPPOVQ2M __mmask8 _mm512_movepi64_mask( __m512i );
VPMPPOVW2M __mmask32 _mm512_movepi16_mask( __m512i );
VPMPPOVB2M __mmask32 _mm256_movepi8_mask( __m256i );
VPMPPOVD2M __mmask8 _mm256_movepi32_mask( __m256i );
VPMPPOVQ2M __mmask8 _mm256_movepi64_mask( __m256i );
VPMPPOVW2M __mmask16 _mm256_movepi16_mask( __m256i );
VPMPPOVB2M __mmask16 _mm_movepi8_mask( __m128i );
VPMPPOVD2M __mmask8 _mm_movepi32_mask( __m128i );
VPMPPOVQ2M __mmask8 _mm_movepi64_mask( __m128i );
VPMPPOVW2M __mmask8 _mm_movepi16_mask( __m128i );

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                    If EVEX.vvvv != 1111B.

## VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F38.W0 32 /r<br>VPMOVQB <i>xmm1/m16</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>   | OVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed byte integers in <i>xmm1/m16</i> with truncation under writemask <i>k1</i> .                             |
| EVEX.128.F3.0F38.W0 22 /r<br>VPMOVSQB <i>xmm1/m16</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>  | OVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed byte integers in <i>xmm1/m16</i> using signed saturation under writemask <i>k1</i> .       |
| EVEX.128.F3.0F38.W0 12 /r<br>VPMOVUSQB <i>xmm1/m16</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> | OVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned byte integers in <i>xmm1/m16</i> using unsigned saturation under writemask <i>k1</i> . |
| EVEX.256.F3.0F38.W0 32 /r<br>VPMOVQB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>   | OVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .                             |
| EVEX.256.F3.0F38.W0 22 /r<br>VPMOVSQB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>  | OVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .       |
| EVEX.256.F3.0F38.W0 12 /r<br>VPMOVUSQB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> | OVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> . |
| EVEX.512.F3.0F38.W0 32 /r<br>VPMOVQB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>   | OVM        | V/V                          | AVX512F                  | Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .                             |
| EVEX.512.F3.0F38.W0 22 /r<br>VPMOVSQB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>  | OVM        | V/V                          | AVX512F                  | Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .       |
| EVEX.512.F3.0F38.W0 12 /r<br>VPMOVUSQB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> | OVM        | V/V                          | AVX512F                  | Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| OVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

m ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/8] ← 0;

**VPMOVB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

m ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])

ELSE

\*DEST[i+7:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VPMOVB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

m ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/8] ← 0;



**VPMOVSQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
  ELSE
    *DEST[i+7:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
  ELSE
    IF *merging-masking*      ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking*    ; zeroing-masking
      DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/8] ← 0;

```

**VPMOVUSQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
  ELSE
    *DEST[i+7:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_cvtsepi64_epi8( __m512i a);
VPMOVSQB __m128i _mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB void _mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_cvtusepi64_epi8( __m512i a);
VPMOVUSQB __m128i _mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVUSQB void _mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);

```

VPMOVUSQB \_\_m128i \_\_mm256\_cvtusepi64\_epi8(\_\_m256i a);  
 VPMOVUSQB \_\_m128i \_\_mm256\_mask\_cvtusepi64\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQB \_\_m128i \_\_mm256\_maskz\_cvtusepi64\_epi8(\_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQB void \_\_mm256\_mask\_cvtusepi64\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQB \_\_m128i \_\_mm\_cvtusepi64\_epi8(\_\_m128i a);  
 VPMOVUSQB \_\_m128i \_\_mm\_mask\_cvtusepi64\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQB \_\_m128i \_\_mm\_maskz\_cvtusepi64\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQB void \_\_mm\_mask\_cvtusepi64\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQB \_\_m128i \_\_mm256\_cvtsepi64\_epi8(\_\_m256i a);  
 VPMOVSQB \_\_m128i \_\_mm256\_mask\_cvtsepi64\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQB \_\_m128i \_\_mm256\_maskz\_cvtsepi64\_epi8(\_\_mmask8 k, \_\_m256i b);  
 VPMOVSQB void \_\_mm256\_mask\_cvtsepi64\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQB \_\_m128i \_\_mm\_cvtsepi64\_epi8(\_\_m128i a);  
 VPMOVSQB \_\_m128i \_\_mm\_mask\_cvtsepi64\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQB \_\_m128i \_\_mm\_maskz\_cvtsepi64\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSQB void \_\_mm\_mask\_cvtsepi64\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQB \_\_m128i \_\_mm256\_cvtepi64\_epi8(\_\_m256i a);  
 VPMOVQB \_\_m128i \_\_mm256\_mask\_cvtepi64\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQB \_\_m128i \_\_mm256\_maskz\_cvtepi64\_epi8(\_\_mmask8 k, \_\_m256i b);  
 VPMOVQB void \_\_mm256\_mask\_cvtepi64\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQB \_\_m128i \_\_mm\_cvtepi64\_epi8(\_\_m128i a);  
 VPMOVQB \_\_m128i \_\_mm\_mask\_cvtepi64\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQB \_\_m128i \_\_mm\_maskz\_cvtepi64\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVQB void \_\_mm\_mask\_cvtepi64\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD                      If EVEX.vvvv != 1111B.

## VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 34 /r<br>VPMOVQW <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>       | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed word integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .                              |
| EVEX.128.F3.0F38.W0 24 /r<br>VPMOVSQW <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>      | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .        |
| EVEX.128.F3.0F38.W0 14 /r<br>VPMOVUSQW <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned word integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .  |
| EVEX.256.F3.0F38.W0 34 /r<br>VPMOVQW <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>       | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .                              |
| EVEX.256.F3.0F38.W0 24 /r<br>VPMOVSQW <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>      | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .        |
| EVEX.256.F3.0F38.W0 14 /r<br>VPMOVUSQW <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .  |
| EVEX.512.F3.0F38.W0 34 /r<br>VPMOVQW <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>      | QVM        | V/V                          | AVX512F                  | Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .                             |
| EVEX.512.F3.0F38.W0 24 /r<br>VPMOVSQW <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>     | QVM        | V/V                          | AVX512F                  | Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .       |
| EVEX.512.F3.0F38.W0 14 /r<br>VPMOVUSQW <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> },<br><i>zmm2</i> | QVM        | V/V                          | AVX512F                  | Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| QVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQW __m128i _mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);

```

VPMOVUSQD \_\_m128i \_\_mm256\_cvtusepi64\_epi32(\_\_m256i a);  
 VPMOVUSQD \_\_m128i \_\_mm256\_mask\_cvtusepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD \_\_m128i \_\_mm256\_maskz\_cvtusepi64\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD void \_\_mm256\_mask\_cvtusepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD \_\_m128i \_\_mm\_cvtusepi64\_epi32(\_\_m128i a);  
 VPMOVUSQD \_\_m128i \_\_mm\_mask\_cvtusepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQD \_\_m128i \_\_mm\_maskz\_cvtusepi64\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQD void \_\_mm\_mask\_cvtusepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD \_\_m128i \_\_mm256\_cvtsepi64\_epi32(\_\_m256i a);  
 VPMOVSQD \_\_m128i \_\_mm256\_mask\_cvtsepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD \_\_m128i \_\_mm256\_maskz\_cvtsepi64\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD void \_\_mm256\_mask\_cvtsepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD \_\_m128i \_\_mm\_cvtsepi64\_epi32(\_\_m128i a);  
 VPMOVSQD \_\_m128i \_\_mm\_mask\_cvtsepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD \_\_m128i \_\_mm\_maskz\_cvtsepi64\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD void \_\_mm\_mask\_cvtsepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD \_\_m128i \_\_mm256\_cvtepi64\_epi32(\_\_m256i a);  
 VPMOVQD \_\_m128i \_\_mm256\_mask\_cvtepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD \_\_m128i \_\_mm256\_maskz\_cvtepi64\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVQD void \_\_mm256\_mask\_cvtepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD \_\_m128i \_\_mm\_cvtepi64\_epi32(\_\_m128i a);  
 VPMOVQD \_\_m128i \_\_mm\_mask\_cvtepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD \_\_m128i \_\_mm\_maskz\_cvtepi64\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVQD void \_\_mm\_mask\_cvtepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|-----------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 35 /r<br>VPMOVQD <i>xmm1/m128 {k1}{z}, xmm2</i>   | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .                             |
| EVEX.128.F3.0F38.W0 25 /r<br>VPMOVSQD <i>xmm1/m64 {k1}{z}, xmm2</i>   | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed double-word integers in <i>xmm1/m64</i> using signed saturation subject to writemask <i>k1</i> .        |
| EVEX.128.F3.0F38.W0 15 /r<br>VPMOVUSQD <i>xmm1/m64 {k1}{z}, xmm2</i>  | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned double-word integers in <i>xmm1/m64</i> using unsigned saturation subject to writemask <i>k1</i> .  |
| EVEX.256.F3.0F38.W0 35 /r<br>VPMOVQD <i>xmm1/m128 {k1}{z}, ymm2</i>   | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .                             |
| EVEX.256.F3.0F38.W0 25 /r<br>VPMOVSQD <i>xmm1/m128 {k1}{z}, ymm2</i>  | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed double-word integers in <i>xmm1/m128</i> using signed saturation subject to writemask <i>k1</i> .       |
| EVEX.256.F3.0F38.W0 15 /r<br>VPMOVUSQD <i>xmm1/m128 {k1}{z}, ymm2</i> | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned double-word integers in <i>xmm1/m128</i> using unsigned saturation subject to writemask <i>k1</i> . |
| EVEX.512.F3.0F38.W0 35 /r<br>VPMOVQD <i>ymm1/m256 {k1}{z}, zmm2</i>   | HVM        | V/V                         | AVX512F                  | Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed double-word integers in <i>ymm1/m256</i> with truncation subject to writemask <i>k1</i> .                             |
| EVEX.512.F3.0F38.W0 25 /r<br>VPMOVSQD <i>ymm1/m256 {k1}{z}, zmm2</i>  | HVM        | V/V                         | AVX512F                  | Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed double-word integers in <i>ymm1/m256</i> using signed saturation subject to writemask <i>k1</i> .       |
| EVEX.512.F3.0F38.W0 15 /r<br>VPMOVUSQD <i>ymm1/m256 {k1}{z}, zmm2</i> | HVM        | V/V                         | AVX512F                  | Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned double-word integers in <i>ymm1/m256</i> using unsigned saturation subject to writemask <i>k1</i> . |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| HVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed doublewords using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VPMOVQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *zeroing-masking*           ; zeroing-masking
        DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;
```

#### VPMOVQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
```

#### VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;
```



**VPMOVSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQD instruction (EVEX encoded version) reg-reg form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*    ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVUSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQD __m256i __mm512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __mm512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD void __mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_cvtsepi64_epi32(__m512i a);
VPMOVSQD __m256i __mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_maskz_cvtsepi64_epi32(__mmask8 k, __m512i a);
VPMOVSQD void __mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i __mm512_cvtusepi64_epi32(__m512i a);
VPMOVUSQD __m256i __mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i __mm512_maskz_cvtusepi64_epi32(__mmask8 k, __m512i a);
VPMOVUSQD void __mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i __mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i __mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);

```

VPMOVUSQD \_\_m128i \_mm256\_maskz\_cvtusepi64\_epi32( \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD void \_mm256\_mask\_cvtusepi64\_storeu\_epi32(void \* , \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD \_\_m128i \_mm\_cvtusepi64\_epi32(\_\_m128i a);  
 VPMOVUSQD \_\_m128i \_mm\_mask\_cvtusepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQD \_\_m128i \_mm\_maskz\_cvtusepi64\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQD void \_mm\_mask\_cvtusepi64\_storeu\_epi32(void \* , \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD \_\_m128i \_mm256\_cvtsepi64\_epi32(\_\_m256i a);  
 VPMOVSQD \_\_m128i \_mm256\_mask\_cvtsepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD \_\_m128i \_mm256\_maskz\_cvtsepi64\_epi32( \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD void \_mm256\_mask\_cvtsepi64\_storeu\_epi32(void \* , \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD \_\_m128i \_mm\_cvtsepi64\_epi32(\_\_m128i a);  
 VPMOVSQD \_\_m128i \_mm\_mask\_cvtsepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD \_\_m128i \_mm\_maskz\_cvtsepi64\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD void \_mm\_mask\_cvtsepi64\_storeu\_epi32(void \* , \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD \_\_m128i \_mm256\_cvtepi64\_epi32(\_\_m256i a);  
 VPMOVQD \_\_m128i \_mm256\_mask\_cvtepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD \_\_m128i \_mm256\_maskz\_cvtepi64\_epi32( \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD void \_mm256\_mask\_cvtepi64\_storeu\_epi32(void \* , \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD \_\_m128i \_mm\_cvtepi64\_epi32(\_\_m128i a);  
 VPMOVQD \_\_m128i \_mm\_mask\_cvtepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD \_\_m128i \_mm\_maskz\_cvtepi64\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD void \_mm\_mask\_cvtepi64\_storeu\_epi32(void \* , \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD                      If EVEX.vvvv != 1111B.

## VPMOVD/VPMSDB/VPMOVSDB—Down Convert DWord to Byte

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 31 /r<br>VPMOVD <i>xmm1/m32</i> {k1}{z}, <i>xmm2</i>       | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask k1.                                |
| EVEX.128.F3.0F38.W0 21 /r<br>VPMSDB <i>xmm1/m32</i> {k1}{z}, <i>xmm2</i>       | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask k1.          |
| EVEX.128.F3.0F38.W0 11 /r<br>VPMOVSDB <i>xmm1/m32</i> {k1}{z},<br><i>xmm2</i>  | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask k1.    |
| EVEX.256.F3.0F38.W0 31 /r<br>VPMOVD <i>xmm1/m64</i> {k1}{z}, <i>ymm2</i>       | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask k1.                                |
| EVEX.256.F3.0F38.W0 21 /r<br>VPMSDB <i>xmm1/m64</i> {k1}{z}, <i>ymm2</i>       | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask k1.          |
| EVEX.256.F3.0F38.W0 11 /r<br>VPMOVSDB <i>xmm1/m64</i> {k1}{z}, <i>ymm2</i>     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask k1.    |
| EVEX.512.F3.0F38.W0 31 /r<br>VPMOVD <i>xmm1/m128</i> {k1}{z}, <i>zmm2</i>      | QVM        | V/V                          | AVX512F                  | Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed byte integers in <i>xmm1/m128</i> with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 21 /r<br>VPMSDB <i>xmm1/m128</i> {k1}{z}, <i>zmm2</i>      | QVM        | V/V                          | AVX512F                  | Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed byte integers in <i>xmm1/m128</i> using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 11 /r<br>VPMOVSDB <i>xmm1/m128</i> {k1}{z},<br><i>zmm2</i> | QVM        | V/V                          | AVX512F                  | Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned byte integers in <i>xmm1/m128</i> using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| QVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

VPMOVD down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVD B instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVD B instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSDB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVSDB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSDB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVUSDB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVDB __m128i _mm512_cvtepi32_epi8( __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8( __mmask16 k, __m512i a);
VPMOVDB void _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSDDB __m128i _mm512_cvtsepi32_epi8( __m512i a);
VPMOVSDDB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSDDB __m128i _mm512_maskz_cvtsepi32_epi8( __mmask16 k, __m512i a);
VPMOVSDDB void _mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_cvtusepi32_epi8( __m512i a);
VPMOVUSDB __m128i _mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_maskz_cvtusepi32_epi8( __mmask16 k, __m512i a);
VPMOVUSDB void _mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm256_cvtusepi32_epi8(__m256i a);
VPMOVUSDB __m128i _mm256_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);

```

VPMOVUSDB \_\_m128i\_mm256\_maskz\_cvtusepi32\_epi8( \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDB void \_\_mm256\_mask\_cvtusepi32\_storeu\_epi8(void \* , \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDB \_\_m128i\_mm\_cvtusepi32\_epi8(\_\_m128i a);  
 VPMOVUSDB \_\_m128i\_mm\_mask\_cvtusepi32\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDB \_\_m128i\_mm\_maskz\_cvtusepi32\_epi8( \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDB void \_\_mm\_mask\_cvtusepi32\_storeu\_epi8(void \* , \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDB \_\_m128i\_mm256\_cvtsepi32\_epi8(\_\_m256i a);  
 VPMOVUSDB \_\_m128i\_mm256\_mask\_cvtsepi32\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDB \_\_m128i\_mm256\_maskz\_cvtsepi32\_epi8( \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDB void \_\_mm256\_mask\_cvtsepi32\_storeu\_epi8(void \* , \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDB \_\_m128i\_mm\_cvtsepi32\_epi8(\_\_m128i a);  
 VPMOVUSDB \_\_m128i\_mm\_mask\_cvtsepi32\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDB \_\_m128i\_mm\_maskz\_cvtsepi32\_epi8( \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDB void \_\_mm\_mask\_cvtsepi32\_storeu\_epi8(void \* , \_\_mmask8 k, \_\_m128i b);  
 VPMOVDB \_\_m128i\_mm256\_cvtepi32\_epi8(\_\_m256i a);  
 VPMOVDB \_\_m128i\_mm256\_mask\_cvtepi32\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVDB \_\_m128i\_mm256\_maskz\_cvtepi32\_epi8( \_\_mmask8 k, \_\_m256i b);  
 VPMOVDB void \_\_mm256\_mask\_cvtepi32\_storeu\_epi8(void \* , \_\_mmask8 k, \_\_m256i b);  
 VPMOVDB \_\_m128i\_mm\_cvtepi32\_epi8(\_\_m128i a);  
 VPMOVDB \_\_m128i\_mm\_mask\_cvtepi32\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVDB \_\_m128i\_mm\_maskz\_cvtepi32\_epi8( \_\_mmask8 k, \_\_m128i b);  
 VPMOVDB void \_\_mm\_mask\_cvtepi32\_storeu\_epi8(void \* , \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 33 /r<br>VPMOVDW <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>       | HVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask k1.                                |
| EVEX.128.F3.0F38.W0 23 /r<br>VPMOVSDW <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>      | HVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask k1.          |
| EVEX.128.F3.0F38.W0 13 /r<br>VPMOVUSDW <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>     | HVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask k1.    |
| EVEX.256.F3.0F38.W0 33 /r<br>VPMOVDW <i>xmm1/m128</i> {k1}{z}, <i>ymm2</i>      | HVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask k1.                               |
| EVEX.256.F3.0F38.W0 23 /r<br>VPMOVSDW <i>xmm1/m128</i> {k1}{z}, <i>ymm2</i>     | HVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask k1.         |
| EVEX.256.F3.0F38.W0 13 /r<br>VPMOVUSDW <i>xmm1/m128</i> {k1}{z},<br><i>ymm2</i> | HVM        | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask k1.   |
| EVEX.512.F3.0F38.W0 33 /r<br>VPMOVDW <i>ymm1/m256</i> {k1}{z}, <i>zmm2</i>      | HVM        | V/V                          | AVX512F                  | Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed word integers in <i>ymm1/m256</i> with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 23 /r<br>VPMOVSDW <i>ymm1/m256</i> {k1}{z}, <i>zmm2</i>     | HVM        | V/V                          | AVX512F                  | Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed word integers in <i>ymm1/m256</i> using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 13 /r<br>VPMOVUSDW <i>ymm1/m256</i> {k1}{z},<br><i>zmm2</i> | HVM        | V/V                          | AVX512F                  | Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned word integers in <i>ymm1/m256</i> using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| HVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

### Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVDW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```



**VPMOVSdW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVUSDW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVDW __m256i __mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i __mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void __mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSdW __m256i __mm512_cvtsepi32_epi16(__m512i a);
VPMOVSdW __m256i __mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSdW __m256i __mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSdW void __mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i __mm512_cvtusepi32_epi16(__m512i a);
VPMOVUSDW __m256i __mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i __mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void __mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);

```

VPMOVUSDW \_\_m128i \_\_mm256\_cvtusepi32\_epi16(\_\_m256i a);  
 VPMOVUSDW \_\_m128i \_\_mm256\_mask\_cvtusepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDW \_\_m128i \_\_mm256\_maskz\_cvtusepi32\_epi16(\_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDW void \_\_mm256\_mask\_cvtusepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDW \_\_m128i \_\_mm\_cvtusepi32\_epi16(\_\_m128i a);  
 VPMOVUSDW \_\_m128i \_\_mm\_mask\_cvtusepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDW \_\_m128i \_\_mm\_maskz\_cvtusepi32\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDW void \_\_mm\_mask\_cvtusepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSDW \_\_m128i \_\_mm256\_cvtsepi32\_epi16(\_\_m256i a);  
 VPMOVSDW \_\_m128i \_\_mm256\_mask\_cvtsepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSDW \_\_m128i \_\_mm256\_maskz\_cvtsepi32\_epi16(\_\_mmask8 k, \_\_m256i b);  
 VPMOVSDW void \_\_mm256\_mask\_cvtsepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSDW \_\_m128i \_\_mm\_cvtsepi32\_epi16(\_\_m128i a);  
 VPMOVSDW \_\_m128i \_\_mm\_mask\_cvtsepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSDW \_\_m128i \_\_mm\_maskz\_cvtsepi32\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSDW void \_\_mm\_mask\_cvtsepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVDW \_\_m128i \_\_mm256\_cvtepi32\_epi16(\_\_m256i a);  
 VPMOVDW \_\_m128i \_\_mm256\_mask\_cvtepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVDW \_\_m128i \_\_mm256\_maskz\_cvtepi32\_epi16(\_\_mmask8 k, \_\_m256i b);  
 VPMOVDW void \_\_mm256\_mask\_cvtepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVDW \_\_m128i \_\_mm\_cvtepi32\_epi16(\_\_m128i a);  
 VPMOVDW \_\_m128i \_\_mm\_mask\_cvtepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVDW \_\_m128i \_\_mm\_maskz\_cvtepi32\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVDW void \_\_mm\_mask\_cvtepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD                    If EVEX.vvvv != 1111B.

## VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F38.W0 30 /r<br>VPMOVWB <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>       | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts 8 packed word integers from <i>xmm2</i> into 8 packed bytes in <i>xmm1/m64</i> with truncation under writemask k1.                                |
| EVEX.128.F3.0F38.W0 20 /r<br>VPMOVSWB <i>xmm1/m64</i> {k1}{z},<br><i>xmm2</i>   | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts 8 packed signed word integers from <i>xmm2</i> into 8 packed signed bytes in <i>xmm1/m64</i> using signed saturation under writemask k1.          |
| EVEX.128.F3.0F38.W0 10 /r<br>VPMOVUSWB <i>xmm1/m64</i> {k1}{z},<br><i>xmm2</i>  | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts 8 packed unsigned word integers from <i>xmm2</i> into 8 packed unsigned bytes in <i>xmm1/m64</i> using unsigned saturation under writemask k1.    |
| EVEX.256.F3.0F38.W0 30 /r<br>VPMOVWB <i>xmm1/m128</i> {k1}{z},<br><i>ymm2</i>   | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts 16 packed word integers from <i>ymm2</i> into 16 packed bytes in <i>xmm1/m128</i> with truncation under writemask k1.                             |
| EVEX.256.F3.0F38.W0 20 /r<br>VPMOVSWB <i>xmm1/m128</i> {k1}{z},<br><i>ymm2</i>  | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts 16 packed signed word integers from <i>ymm2</i> into 16 packed signed bytes in <i>xmm1/m128</i> using signed saturation under writemask k1.       |
| EVEX.256.F3.0F38.W0 10 /r<br>VPMOVUSWB <i>xmm1/m128</i> {k1}{z},<br><i>ymm2</i> | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Converts 16 packed unsigned word integers from <i>ymm2</i> into 16 packed unsigned bytes in <i>xmm1/m128</i> using unsigned saturation under writemask k1. |
| EVEX.512.F3.0F38.W0 30 /r<br>VPMOVWB <i>ymm1/m256</i> {k1}{z},<br><i>zmm2</i>   | HVM        | V/V                          | AVX512BW                 | Converts 32 packed word integers from <i>zmm2</i> into 32 packed bytes in <i>ymm1/m256</i> with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 20 /r<br>VPMOVSWB <i>ymm1/m256</i> {k1}{z},<br><i>zmm2</i>  | HVM        | V/V                          | AVX512BW                 | Converts 32 packed signed word integers from <i>zmm2</i> into 32 packed signed bytes in <i>ymm1/m256</i> using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 10 /r<br>VPMOVUSWB <i>ymm1/m256</i> {k1}{z},<br><i>zmm2</i> | HVM        | V/V                          | AVX512BW                 | Converts 32 packed unsigned word integers from <i>zmm2</i> into 32 packed unsigned bytes in <i>ymm1/m256</i> using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| HVM   | ModRM:r/m (w) | ModRM:reg (r) | NA        | NA        |

#### Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VPMOVB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] = 0
      FI
    FI;
  ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;
```

#### VPMOVB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
  ENDFOR
```

#### VPMOVS instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] = 0
      FI
    FI;
  ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;
```

**VPMOVS WB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUS WB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVUS WB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVUSWB __m256i __mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i __mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i __mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void __mm512_mask_cvtusepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m256i __mm512_cvtsepi16_epi8(__m512i a);
VPMOVUSWB __m256i __mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i __mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void __mm512_mask_cvtsepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVWB __m256i __mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i __mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i __mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVWB void __mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);

```

VPMOVSXB \_\_m128i \_\_mm256\_cvtusepi16\_epi8(\_\_m256i a);  
 VPMOVSXB \_\_m128i \_\_mm256\_mask\_cvtusepi16\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB \_\_m128i \_\_mm256\_maskz\_cvtusepi16\_epi8(\_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB void \_\_mm256\_mask\_cvtusepi16\_storeu\_epi8(void \*, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB \_\_m128i \_\_mm\_cvtusepi16\_epi8(\_\_m128i a);  
 VPMOVSXB \_\_m128i \_\_mm\_mask\_cvtusepi16\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB \_\_m128i \_\_mm\_maskz\_cvtusepi16\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB void \_\_mm\_mask\_cvtusepi16\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB \_\_m128i \_\_mm256\_cvtsepi16\_epi8(\_\_m256i a);  
 VPMOVSXB \_\_m128i \_\_mm256\_mask\_cvtsepi16\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB \_\_m128i \_\_mm256\_maskz\_cvtsepi16\_epi8(\_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB void \_\_mm256\_mask\_cvtsepi16\_storeu\_epi8(void \*, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB \_\_m128i \_\_mm\_cvtsepi16\_epi8(\_\_m128i a);  
 VPMOVSXB \_\_m128i \_\_mm\_mask\_cvtsepi16\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB \_\_m128i \_\_mm\_maskz\_cvtsepi16\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB void \_\_mm\_mask\_cvtsepi16\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB \_\_m128i \_\_mm256\_cvtepi16\_epi8(\_\_m256i a);  
 VPMOVSXB \_\_m128i \_\_mm256\_mask\_cvtepi16\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB \_\_m128i \_\_mm256\_maskz\_cvtepi16\_epi8(\_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB void \_\_mm256\_mask\_cvtepi16\_storeu\_epi8(void \*, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXB \_\_m128i \_\_mm\_cvtepi16\_epi8(\_\_m128i a);  
 VPMOVSXB \_\_m128i \_\_mm\_mask\_cvtepi16\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB \_\_m128i \_\_mm\_maskz\_cvtepi16\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSXB void \_\_mm\_mask\_cvtepi16\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF

#UD If EVEX.vvvv != 1111B.

## PMOVSX—Packed Move with Sign Extend

| Opcode/<br>Instruction                                 | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| 66 0f 38 20 /r<br>PMOVSXBW xmm1, xmm2/m64              | RM        | V/V                          | SSE4_1                   | Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.    |
| 66 0f 38 21 /r<br>PMOVSXBD xmm1, xmm2/m32              | RM        | V/V                          | SSE4_1                   | Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.    |
| 66 0f 38 22 /r<br>PMOVSXBQ xmm1, xmm2/m16              | RM        | V/V                          | SSE4_1                   | Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.    |
| 66 0f 38 23/r<br>PMOVSXWD xmm1, xmm2/m64               | RM        | V/V                          | SSE4_1                   | Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.   |
| 66 0f 38 24 /r<br>PMOVSXWQ xmm1, xmm2/m32              | RM        | V/V                          | SSE4_1                   | Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.   |
| 66 0f 38 25 /r<br>PMOVSXDQ xmm1, xmm2/m64              | RM        | V/V                          | SSE4_1                   | Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.   |
| VEX.128.66.0F38.WIG 20 /r<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.256.66.0F38.WIG 24 /r<br>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<br>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.                          |
| EVEX.128.66.0F38.WIG 20 /r<br>VPMOVSXBW xmm1 {k1}{z},<br>xmm2/m64  | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.  |
| EVEX.256.66.0F38.WIG 20 /r<br>VPMOVSXBW ymm1 {k1}{z},<br>xmm2/m128 | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.   |
| EVEX.512.66.0F38.WIG 20 /r<br>VPMOVSXBW zmm1 {k1}{z},<br>ymm2/m256 | HVM        | V/V                          | AVX512BW                 | Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.   |
| EVEX.128.66.0F38.WIG 21 /r<br>VPMOVSXBD xmm1 {k1}{z},<br>xmm2/m32  | QVM        | V/V                          | AVX512VL<br>AVX512F      | Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.     |
| EVEX.256.66.0F38.WIG 21 /r<br>VPMOVSXBD ymm1 {k1}{z},<br>xmm2/m64  | QVM        | V/V                          | AVX512VL<br>AVX512F      | Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.     |
| EVEX.512.66.0F38.WIG 21 /r<br>VPMOVSXBD zmm1 {k1}{z},<br>xmm2/m128 | QVM        | V/V                          | AVX512F                  | Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1. |
| EVEX.128.66.0F38.WIG 22 /r<br>VPMOVSXBQ xmm1 {k1}{z},<br>xmm2/m16  | OVM        | V/V                          | AVX512VL<br>AVX512F      | Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.     |
| EVEX.256.66.0F38.WIG 22 /r<br>VPMOVSXBQ ymm1 {k1}{z},<br>xmm2/m32  | OVM        | V/V                          | AVX512VL<br>AVX512F      | Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.     |
| EVEX.512.66.0F38.WIG 22 /r<br>VPMOVSXBQ zmm1 {k1}{z},<br>xmm2/m64  | OVM        | V/V                          | AVX512F                  | Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.     |
| EVEX.128.66.0F38.WIG 23 /r<br>VPMOVSXWD xmm1 {k1}{z},<br>xmm2/m64  | HVM        | V/V                          | AVX512VL<br>AVX512F      | Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.WIG 23 /r<br>VPMOVSXWD ymm1 {k1}{z},<br>xmm2/m128 | HVM        | V/V                          | AVX512VL<br>AVX512F      | Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.  |



| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.512.66.0F38.WIG 23 /r<br>VPMOVSXWD zmm1 {k1}{z},<br>ymm2/m256 | HVM       | V/V                          | AVX512F                  | Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1. |
| EVEX.128.66.0F38.WIG 24 /r<br>VPMOVSXWQ xmm1 {k1}{z},<br>xmm2/m32  | QVM       | V/V                          | AVX512VL<br>AVX512F      | Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.     |
| EVEX.256.66.0F38.WIG 24 /r<br>VPMOVSXWQ ymm1 {k1}{z},<br>xmm2/m64  | QVM       | V/V                          | AVX512VL<br>AVX512F      | Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.     |
| EVEX.512.66.0F38.WIG 24 /r<br>VPMOVSXWQ zmm1 {k1}{z},<br>xmm2/m128 | QVM       | V/V                          | AVX512F                  | Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.   |
| EVEX.128.66.0F38.W0 25 /r<br>VPMOVSDQ xmm1 {k1}{z},<br>xmm2/m64    | HVM       | V/V                          | AVX512VL<br>AVX512F      | Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.          |
| EVEX.256.66.0F38.W0 25 /r<br>VPMOVSDQ ymm1 {k1}{z},<br>xmm2/m128   | HVM       | V/V                          | AVX512VL<br>AVX512F      | Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.        |
| EVEX.512.66.0F38.W0 25 /r<br>VPMOVSDQ zmm1 {k1}{z},<br>ymm2/m256   | HVM       | V/V                          | AVX512F                  | Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.        |

### Instruction Operand Encoding

| Op/En         | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|---------------|---------------|---------------|-----------|-----------|
| RM            | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| HVM, QVM, OVM | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation****Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST, SRC)**

```

DEST[15:0] ← SignExtend(SRC[7:0]);
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);

```

**Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST, SRC)**

```

DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);

```

**Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST, SRC)**

```

DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);

```

**Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST, SRC)**

```

DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);

```

**Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST, SRC)**

```

DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);

```

**Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST, SRC)**

```

DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);

```

**VPMOVSXBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[127:0], SRC[63:0])

IF VL >= 256

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL >= 512

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TEMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0

```

        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVSXBD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[383:256], SRC[95:64])

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TEMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXBQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[127:0], SRC[15:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[255:128], SRC[31:16])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[383:256], SRC[47:32])

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[511:384], SRC[63:48])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXWD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[511:384], SRC[256:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TEMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXWQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[383:256], SRC[95:64])

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXBW (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[255:128], SRC[127:64])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXBD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[31:0])

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[255:128], SRC[63:32])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXBQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[15:0])

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[255:128], SRC[31:16])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXWD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[255:128], SRC[127:64])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXWQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[31:0])

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[255:128], SRC[63:32])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXDQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXBW (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_WORDDEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXBD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXBQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXWD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXWQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXDQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**PMOVSXBW**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXBD**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXBQ**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXWD**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXWQ**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXDQ**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMOVSXBW \_\_m512i \_\_mm512\_cvtepi8\_epi16(\_\_m512i a);  
 VPMOVSXBW \_\_m512i \_\_mm512\_mask\_cvtepi8\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i b);  
 VPMOVSXBW \_\_m512i \_\_mm512\_maskz\_cvtepi8\_epi16(\_\_mmask32 k, \_\_m512i b);  
 VPMOVSXBD \_\_m512i \_\_mm512\_cvtepi8\_epi32(\_\_m512i a);  
 VPMOVSXBD \_\_m512i \_\_mm512\_mask\_cvtepi8\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b);

VPMOVSXBD \_\_m512i \_mm512\_maskz\_cvtepi8\_epi32( \_\_mmask16 k, \_\_m512i b);  
 VPMOVSXBQ \_\_m512i \_mm512\_cvtepi8\_epi64(\_\_m512i a);  
 VPMOVSXBQ \_\_m512i \_mm512\_mask\_cvtepi8\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSXBQ \_\_m512i \_mm512\_maskz\_cvtepi8\_epi64( \_\_mmask8 k, \_\_m512i a);  
 VPMOVSXDQ \_\_m512i \_mm512\_cvtepi32\_epi64(\_\_m512i a);  
 VPMOVSXDQ \_\_m512i \_mm512\_mask\_cvtepi32\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSXDQ \_\_m512i \_mm512\_maskz\_cvtepi32\_epi64( \_\_mmask8 k, \_\_m512i a);  
 VPMOVSXWD \_\_m512i \_mm512\_cvtepi16\_epi32(\_\_m512i a);  
 VPMOVSXWD \_\_m512i \_mm512\_mask\_cvtepi16\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b);  
 VPMOVSXWD \_\_m512i \_mm512\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m512i a);  
 VPMOVSXWQ \_\_m512i \_mm512\_cvtepi16\_epi64(\_\_m512i a);  
 VPMOVSXWQ \_\_m512i \_mm512\_mask\_cvtepi16\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSXWQ \_\_m512i \_mm512\_maskz\_cvtepi16\_epi64( \_\_mmask8 k, \_\_m512i a);  
 VPMOVSXBW \_\_m256i \_mm256\_cvtepi8\_epi16(\_\_m256i a);  
 VPMOVSXBW \_\_m256i \_mm256\_mask\_cvtepi8\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXBW \_\_m256i \_mm256\_maskz\_cvtepi8\_epi16( \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXBD \_\_m256i \_mm256\_cvtepi8\_epi32(\_\_m256i a);  
 VPMOVSXBD \_\_m256i \_mm256\_mask\_cvtepi8\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXBD \_\_m256i \_mm256\_maskz\_cvtepi8\_epi32( \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXBQ \_\_m256i \_mm256\_cvtepi8\_epi64(\_\_m256i a);  
 VPMOVSXBQ \_\_m256i \_mm256\_mask\_cvtepi8\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXBQ \_\_m256i \_mm256\_maskz\_cvtepi8\_epi64( \_\_mmask8 k, \_\_m256i a);  
 VPMOVSXDQ \_\_m256i \_mm256\_cvtepi32\_epi64(\_\_m256i a);  
 VPMOVSXDQ \_\_m256i \_mm256\_mask\_cvtepi32\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXDQ \_\_m256i \_mm256\_maskz\_cvtepi32\_epi64( \_\_mmask8 k, \_\_m256i a);  
 VPMOVSXWD \_\_m256i \_mm256\_cvtepi16\_epi32(\_\_m256i a);  
 VPMOVSXWD \_\_m256i \_mm256\_mask\_cvtepi16\_epi32(\_\_m256i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXWD \_\_m256i \_mm256\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m256i a);  
 VPMOVSXWQ \_\_m256i \_mm256\_cvtepi16\_epi64(\_\_m256i a);  
 VPMOVSXWQ \_\_m256i \_mm256\_mask\_cvtepi16\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXWQ \_\_m256i \_mm256\_maskz\_cvtepi16\_epi64( \_\_mmask8 k, \_\_m256i a);  
 VPMOVSXBW \_\_m128i \_mm\_mask\_cvtepi8\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBW \_\_m128i \_mm\_maskz\_cvtepi8\_epi16( \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBD \_\_m128i \_mm\_mask\_cvtepi8\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBD \_\_m128i \_mm\_maskz\_cvtepi8\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBQ \_\_m128i \_mm\_mask\_cvtepi8\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBQ \_\_m128i \_mm\_maskz\_cvtepi8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVSXDQ \_\_m128i \_mm\_mask\_cvtepi32\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXDQ \_\_m128i \_mm\_maskz\_cvtepi32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVSXWD \_\_m128i \_mm\_mask\_cvtepi16\_epi32(\_\_m128i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVSXWD \_\_m128i \_mm\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVSXWQ \_\_m128i \_mm\_mask\_cvtepi16\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXWQ \_\_m128i \_mm\_maskz\_cvtepi16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 PMOVSXBW \_\_m128i \_mm\_cvtepi8\_epi16 ( \_\_m128i a);  
 PMOVSXBD \_\_m128i \_mm\_cvtepi8\_epi32 ( \_\_m128i a);  
 PMOVSXBQ \_\_m128i \_mm\_cvtepi8\_epi64 ( \_\_m128i a);  
 PMOVSXWD \_\_m128i \_mm\_cvtepi16\_epi32 ( \_\_m128i a);  
 PMOVSXWQ \_\_m128i \_mm\_cvtepi16\_epi64 ( \_\_m128i a);  
 PMOVSXDQ \_\_m128i \_mm\_cvtepi32\_epi64 ( \_\_m128i a);

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD                      If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.



## PMOVZX—Packed Move with Zero Extend

| Opcode/<br>Instruction                                 | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0f 38 30 /r<br>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<br>PMOVZXBW 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<br>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<br>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<br>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<br>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<br>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<br>VPMOVZXBW 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<br>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<br>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<br>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.128.66.0F 38.WIG 35 /r<br>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<br>VPMOVZXBW ymm1, xmm2/m128 | RM         | V/V                          | AVX2                     | Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.                  |
| VEX.256.66.0F38.WIG 31 /r<br>VPMOVZXBW 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<br>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<br>VPMOVZXWD ymm1, xmm2/m128 | RM         | V/V                          | AVX2                     | Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.                      |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| VEX.256.66.0F38.WIG 34 /r<br>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<br>VPMOVZXDQ ymm1, xmm2/m128             | RM         | V/V                          | AVX2                     | Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.   |
| EVEX.128.66.0F38.30.WIG /r<br>VPMOVZXBW xmm1 {k1}{z}, xmm2/m64     | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.                          |
| EVEX.256.66.0F38.WIG 30 /r<br>VPMOVZXBW ymm1 {k1}{z},<br>xmm2/m128 | HVM        | V/V                          | AVX512VL<br>AVX512BW     | Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.  |
| EVEX.512.66.0F38.WIG 30 /r<br>VPMOVZXBW zmm1 {k1}{z},<br>ymm2/m256 | HVM        | V/V                          | AVX512BW                 | Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.  |
| EVEX.128.66.0F38.WIG 31 /r<br>VPMOVZXBD xmm1 {k1}{z}, xmm2/m32     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.  |
| EVEX.256.66.0F38.WIG 31 /r<br>VPMOVZXBD ymm1 {k1}{z}, xmm2/m64     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.  |
| EVEX.512.66.0F38.WIG 31 /r<br>VPMOVZXBD zmm1 {k1}{z},<br>xmm2/m128 | QVM        | V/V                          | AVX512F                  | Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.                  |
| EVEX.128.66.0F38.WIG 32 /r<br>VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16     | OVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.  |
| EVEX.256.66.0F38.WIG 32 /r<br>VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32     | OVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.  |
| EVEX.512.66.0F38.WIG 32 /r<br>VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64     | OVM        | V/V                          | AVX512F                  | Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.  |
| EVEX.128.66.0F38.WIG 33 /r<br>VPMOVZXWD xmm1 {k1}{z}, xmm2/m64     | HVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1. |
| EVEX.256.66.0F38.WIG 33 /r<br>VPMOVZXWD ymm1 {k1}{z},<br>xmm2/m128 | HVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.                   |
| EVEX.512.66.0F38.WIG 33 /r<br>VPMOVZXWD zmm1 {k1}{z},<br>ymm2/m256 | HVM        | V/V                          | AVX512F                  | Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.                 |
| EVEX.128.66.0F38.WIG 34 /r<br>VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| EVEX.256.66.0F38.WIG 34 /r<br>VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64     | QVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1. |
| EVEX.512.66.0F38.WIG 34 /r<br>VPMOVZXWQ zmm1 {k1}{z},<br>xmm2/m128 | QVM        | V/V                          | AVX512F                  | Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.                   |
| EVEX.128.66.0F38.W0 35 /r<br>VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64      | HVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.      |
| EVEX.256.66.0F38.W0 35 /r<br>VPMOVZXDQ ymm1 {k1}{z},<br>xmm2/m128  | HVM        | V/V                          | AVX512VL<br>AVX512F      | Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.                        |
| EVEX.512.66.0F38.W0 35 /r<br>VPMOVZXDQ zmm1 {k1}{z},<br>ymm2/m256  | HVM        | V/V                          | AVX512F                  | Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.                        |

### Instruction Operand Encoding

| Op/En         | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|---------------|---------------|---------------|-----------|-----------|
| RM            | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| HVM, QVM, OVM | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Legacy, VEX and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST, SRC)

```
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);
```

#### Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST, SRC)

```
DEST[31:0] ← ZeroExtend(SRC[7:0]);
```

DEST[63:32] ← ZeroExtend(SRC[15:8]);  
 DEST[95:64] ← ZeroExtend(SRC[23:16]);  
 DEST[127:96] ← ZeroExtend(SRC[31:24]);

**Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[7:0]);  
 DEST[127:64] ← ZeroExtend(SRC[15:8]);

**Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST, SRC)**

DEST[31:0] ← ZeroExtend(SRC[15:0]);  
 DEST[63:32] ← ZeroExtend(SRC[31:16]);  
 DEST[95:64] ← ZeroExtend(SRC[47:32]);  
 DEST[127:96] ← ZeroExtend(SRC[63:48]);

**Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[15:0]);  
 DEST[127:64] ← ZeroExtend(SRC[31:16]);

**Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[31:0]);  
 DEST[127:64] ← ZeroExtend(SRC[63:32]);

**VPMOVZXBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[127:0], SRC[63:0])

IF VL >= 256

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL >= 512

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[383:256], SRC[191:128])

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

    i ← j \* 16

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+15:i] ← TEMP\_DEST[i+15:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+15:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+15:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVZXBW (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL >= 256

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL >= 512

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[383:256], SRC[95:64])

```

Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXBQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXWD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]

```

```

ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXWQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL &gt;= 256

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL &gt;= 512

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[383:256], SRC[95:64])

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[j+63:i] ← TEMP\_DEST[j+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVZXDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[383:256], SRC[191:128])

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[j+63:i] ← TEMP\_DEST[j+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXBW (VEX.256 encoded version)**

```

Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAX_VL-1:256] ← 0

```

**VPMOVZXBW (VEX.256 encoded version)**

```

Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAX_VL-1:256] ← 0

```

**VPMOVZXBQ (VEX.256 encoded version)**

```

Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAX_VL-1:256] ← 0

```

**VPMOVZXWD (VEX.256 encoded version)**

```

Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAX_VL-1:256] ← 0

```

**VPMOVZXWQ (VEX.256 encoded version)**

```

Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAX_VL-1:256] ← 0

```

**VPMOVZXDQ (VEX.256 encoded version)**

```

Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAX_VL-1:256] ← 0

```

**VPMOVZXBW (VEX.128 encoded version)**

```

Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAX_VL-1:128] ← 0

```

**VPMOVZXBW (VEX.128 encoded version)**

```

Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAX_VL-1:128] ← 0

```

**VPMOVZXBQ (VEX.128 encoded version)**

```

Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAX_VL-1:128] ← 0

```

**VPMOVZXWD (VEX.128 encoded version)**

```

Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAX_VL-1:128] ← 0

```

**VPMOVZXWQ (VEX.128 encoded version)**

```

Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAX_VL-1:128] ← 0

```

**VPMOVZXDQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD()

DEST[MAX\_VL-1:128] ← 0

**PMOVZXBW**

Packed\_Zero\_Extend\_BYTE\_to\_WORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXBQ**

Packed\_Zero\_Extend\_BYTE\_to\_DWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXBQ**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXWD**

Packed\_Zero\_Extend\_WORD\_to\_DWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXWQ**

Packed\_Zero\_Extend\_WORD\_to\_QWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXDQ**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMOVZXBW \_\_m512i \_\_mm512\_cvtepu8\_epi16(\_\_m256i a);  
 VPMOVZXBW \_\_m512i \_\_mm512\_mask\_cvtepu8\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m256i b);  
 VPMOVZXBW \_\_m512i \_\_mm512\_maskz\_cvtepu8\_epi16(\_\_mmask32 k, \_\_m256i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_cvtepu8\_epi32(\_\_m128i a);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_mask\_cvtepu8\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_maskz\_cvtepu8\_epi32(\_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_cvtepu8\_epi64(\_\_m128i a);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_mask\_cvtepu8\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_maskz\_cvtepu8\_epi64(\_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m512i \_\_mm512\_cvtepu32\_epi64(\_\_m256i a);  
 VPMOVZXDQ \_\_m512i \_\_mm512\_mask\_cvtepu32\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVZXDQ \_\_m512i \_\_mm512\_maskz\_cvtepu32\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVZXWD \_\_m512i \_\_mm512\_cvtepu16\_epi32(\_\_m128i a);  
 VPMOVZXWD \_\_m512i \_\_mm512\_mask\_cvtepu16\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m512i \_\_mm512\_maskz\_cvtepu16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m512i \_\_mm512\_cvtepu16\_epi64(\_\_m256i a);  
 VPMOVZXWQ \_\_m512i \_\_mm512\_mask\_cvtepu16\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVZXWQ \_\_m512i \_\_mm512\_maskz\_cvtepu16\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVZXBW \_\_m256i \_\_mm256\_cvtepu8\_epi16(\_\_m256i a);  
 VPMOVZXBW \_\_m256i \_\_mm256\_mask\_cvtepu8\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBW \_\_m256i \_\_mm256\_maskz\_cvtepu8\_epi16(\_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_cvtepu8\_epi32(\_\_m128i a);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_mask\_cvtepu8\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_maskz\_cvtepu8\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_cvtepu8\_epi64(\_\_m128i a);



VPMOVZXBQ \_\_m256i \_\_mm256\_mask\_cvtepu8\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_maskz\_cvtepu8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m256i \_\_mm256\_cvtepu32\_epi64(\_\_m128i a);  
 VPMOVZXDQ \_\_m256i \_\_mm256\_mask\_cvtepu32\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXDQ \_\_m256i \_\_mm256\_maskz\_cvtepu32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWD \_\_m256i \_\_mm256\_cvtepu16\_epi32(\_\_m128i a);  
 VPMOVZXWD \_\_m256i \_\_mm256\_mask\_cvtepu16\_epi32(\_\_m256i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m256i \_\_mm256\_maskz\_cvtepu16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_cvtepu16\_epi64(\_\_m128i a);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_mask\_cvtepu16\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_maskz\_cvtepu16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXBW \_\_m128i \_\_mm\_mask\_cvtepu8\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi16( \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_mask\_cvtepu8\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m128i \_\_mm\_mask\_cvtepu8\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m128i \_\_mm\_mask\_cvtepu32\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXDQ \_\_m128i \_\_mm\_maskz\_cvtepu32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWD \_\_m128i \_\_mm\_mask\_cvtepu16\_epi32(\_\_m128i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m128i \_\_mm\_maskz\_cvtepu16\_epi32(\_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m128i \_\_mm\_mask\_cvtepu16\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXWQ \_\_m128i \_\_mm\_maskz\_cvtepu16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 PMOVZXBW \_\_m128i \_\_mm\_ cvtepu8\_epi16 ( \_\_m128i a);  
 PMOVZXBW \_\_m128i \_\_mm\_ cvtepu8\_epi32 ( \_\_m128i a);  
 PMOVZXBQ \_\_m128i \_\_mm\_ cvtepu8\_epi64 ( \_\_m128i a);  
 PMOVZXWD \_\_m128i \_\_mm\_ cvtepu16\_epi32 ( \_\_m128i a);  
 PMOVZXWQ \_\_m128i \_\_mm\_ cvtepu16\_epi64 ( \_\_m128i a);  
 PMOVZXDQ \_\_m128i \_\_mm\_ cvtepu32\_epi64 ( \_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PMULDQ—Multiply Packed Doubleword Integers

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F 38 28 /r<br>PMULDQ xmm1, xmm2/m128  | RM         | V/V                          | SSE4_1                   | Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.                            |
| VEX.NDS.128.66.0F38.WIG 28 /r<br>VPMULDQ xmm1, xmm2,<br>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<br>VPMULDQ ymm1, ymm2,<br>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.                            |
| EVEX.NDS.128.66.0F38.W1 28 /r<br>VPMULDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F38.W1 28 /r<br>VPMULDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F38.W1 28 /r<br>VPMULDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source

operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

### Operation

#### VPMULDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+63:i] ← SignExtend64( SRC1[i+31:i] ) \* SignExtend64( SRC2[31:0] )

        ELSE DEST[i+63:i] ← SignExtend64( SRC1[j+31:i] ) \* SignExtend64( SRC2[i+31:i] )

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

          DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VPMULDQ (VEX.256 encoded version)

DEST[63:0] ← SignExtend64( SRC1[31:0] ) \* SignExtend64( SRC2[31:0] )

DEST[127:64] ← SignExtend64( SRC1[95:64] ) \* SignExtend64( SRC2[95:64] )

DEST[191:128] ← SignExtend64( SRC1[159:128] ) \* SignExtend64( SRC2[159:128] )

DEST[255:192] ← SignExtend64( SRC1[223:192] ) \* SignExtend64( SRC2[223:192] )

DEST[MAX\_VL-1:256] ← 0

#### VPMULDQ (VEX.128 encoded version)

DEST[63:0] ← SignExtend64( SRC1[31:0] ) \* SignExtend64( SRC2[31:0] )

DEST[127:64] ← SignExtend64( SRC1[95:64] ) \* SignExtend64( SRC2[95:64] )

DEST[MAX\_VL-1:128] ← 0

#### PMULDQ (128-bit Legacy SSE version)

DEST[63:0] ← SignExtend64( DEST[31:0] ) \* SignExtend64( SRC[31:0] )

DEST[127:64] ← SignExtend64( DEST[95:64] ) \* SignExtend64( SRC[95:64] )

DEST[MAX\_VL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

VPMULDQ \_\_m512i \_\_mm512\_mul\_epi32(\_\_m512i a, \_\_m512i b);

VPMULDQ \_\_m512i \_\_mm512\_mask\_mul\_epi32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULDQ \_\_m512i \_\_mm512\_maskz\_mul\_epi32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULDQ \_\_m256i \_\_mm256\_mask\_mul\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULDQ \_\_m256i \_\_mm256\_mask\_mul\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULDQ \_\_m128i \_\_mm\_mask\_mul\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULDQ \_\_m128i \_\_mm\_mask\_mul\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMULDQ \_\_m128i \_mm\_mul\_epi32(\_\_m128i a, \_\_m128i b);  
VPMULDQ \_\_m256i \_mm256\_mul\_epi32(\_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMULHRWSW—Multiply Packed Unsigned Integers with Round and Scale

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| 66 0F 38 0B /r<br>PMULHRWSW 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 0B /r<br>VPMULHRWSW 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 0B /r<br>VPMULHRWSW ymm1, ymm2, ymm3/m256                 | RVM       | V/V                          | AVX2                     | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1.                    |
| EVEX.NDS.128.66.0F38.WIG 0B /r<br>VPMULHRWSW xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.WIG 0B /r<br>VPMULHRWSW ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.WIG 0B /r<br>VPMULHRWSW zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

PMULHRWSW multiplies vertically each signed 16-bit integer from the first source operand with the corresponding signed 16-bit integer of the second source 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.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The second source operand can be a vector register or a memory location. The first source and destination operands are vector registers.

### Operation

#### VPMULHRWSW (EVEX encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

  i ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN

      temp[31:0] ← ((SRC1[j+15:i] \* SRC2[j+15:i]) >>14) + 1

      DEST[j+15:i] ← tmp[16:1]

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[j+15:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

          DEST[j+15:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VPMULHRW (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[175:160] \* SRC2[175:160]) >>14) + 1

temp11[31:0] ← INT32 ((SRC1[191:176] \* SRC2[191:176]) >>14) + 1

temp12[31:0] ← INT32 ((SRC1[207:192] \* SRC2[207:192]) >>14) + 1

temp13[31:0] ← INT32 ((SRC1[223:208] \* SRC2[223:208]) >>14) + 1

temp14[31:0] ← INT32 ((SRC1[239:224] \* SRC2[239:224]) >>14) + 1

temp15[31:0] ← INT32 ((SRC1[255:240] \* SRC2[255:240]) >>14) + 1

DEST[15:0] ← temp0[16:1]

DEST[31:16] ← temp1[16:1]

DEST[47:32] ← temp2[16:1]

DEST[63:48] ← temp3[16:1]

DEST[79:64] ← temp4[16:1]

DEST[95:80] ← temp5[16:1]

DEST[111:96] ← temp6[16:1]

DEST[127:112] ← temp7[16:1]

DEST[143:128] ← temp8[16:1]

DEST[159:144] ← temp9[16:1]

DEST[175:160] ← temp10[16:1]

DEST[191:176] ← temp11[16:1]

DEST[207:192] ← temp12[16:1]

DEST[223:208] ← temp13[16:1]

DEST[239:224] ← temp14[16:1]

DEST[255:240] ← temp15[16:1]

DEST[MAX\_VL-1:256] ← 0

### VPMULHRW (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[MAX_VL-1:128] ← 0

```

### PMULHRSW (128-bit Legacy SSE version)

```

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]
DEST[MAX_VL-1:128] (Unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPMULHRSW __m512i __mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i __mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i __mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i __mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i __mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i __mm128_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i __mm128_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
(V)PMULHRSW __m128i __mm128i_mulhrs_epi16(__m128i a, __m128i b);
VPMULHRSW __m256i __mm256i_mulhrs_epi16(__m256i a, __m256i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULHUW—Multiply Packed Unsigned Integers and Store High Result

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F E4 /r<br>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 E4 /r<br>VPMULHUW xmm1, xmm2,<br>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 E4 /r<br>VPMULHUW ymm1, ymm2,<br>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.                    |
| EVEX.NDS.128.66.0F.WIG E4 /r<br>VPMULHUW xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG E4 /r<br>VPMULHUW ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.WIG E4 /r<br>VPMULHUW zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the first source operand and the second source operand, and stores the high 16 bits of each 32-bit intermediate results 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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The second source operand can be a vector register or a memory location. The first source and destination operands are vector registers.

### Operation

**PMULHUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR  $j \leftarrow 0$  TO KL-1



```

i ← j * 16
IF k1[j] OR *no writemask*
  THEN
    temp[31:0] ← SRC1[j+15:i] * SRC2[j+15:i]
    DEST[j+15:i] ← tmp[31:16]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[j+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**PMULHUW (VEX.256 encoded version)**

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]

```

```

DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAX_VL-1:256] ← 0

```

**PMULHUW (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[MAX\_VL-1:128] ← 0

**PMULHUW (128-bit Legacy SSE version)**

TEMP0[31:0] ← DEST[15:0] \* SRC[15:0]  
 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]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULHUW \_\_m512i \_\_mm512\_mulhi\_epu16(\_\_m512i a, \_\_m512i b);  
 VPMULHUW \_\_m512i \_\_mm512\_mask\_mulhi\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHUW \_\_m512i \_\_mm512\_maskz\_mulhi\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHUW \_\_m256i \_\_mm256\_mask\_mulhi\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHUW \_\_m256i \_\_mm256\_maskz\_mulhi\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHUW \_\_m128i \_\_mm\_mask\_mulhi\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULHUW \_\_m128i \_\_mm\_maskz\_mulhi\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PMULHUW \_\_m128i \_\_mm\_mulhi\_epu16 (\_\_m128i a, \_\_m128i b)  
 VPMULHUW \_\_m256i \_\_mm256\_mulhi\_epu16 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULHW—Multiply Packed Integers and Store High Result

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F E5 /r<br>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 E5 /r<br>VPMULHW xmm1, xmm2,<br>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 E5 /r<br>VPMULHW ymm1, ymm2,<br>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.                    |
| EVEX.NDS.128.66.0F.WIG E5 /r<br>VPMULHW xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG E5 /r<br>VPMULHW ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.WIG E5 /r<br>VPMULHW zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD signed multiply of the packed signed word integers in the first source operand and the second source operand, and stores the high 16 bits of each intermediate 32-bit result 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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The second source operand can be a vector register or a memory location. The first source and destination operands are vector registers.

### Operation

#### PMULHW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR  $j \leftarrow 0$  TO KL-1

```

i ← j * 16
IF k1[j] OR *no writemask*
  THEN
    temp[31:0] ← SRC1[j+15:i] * SRC2[j+15:i]
    DEST[j+15:i] ← tmp[31:16]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[j+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**PMULHW (VEX.256 encoded version)**

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]

```

```

DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAX_VL-1:256] ← 0

```

**PMULHW (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[MAX\_VL-1:128] ← 0

#### PMULHW (128-bit Legacy SSE version)

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]  
 DEST[MAX\_VL-1:128] (Unmodified)

#### Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW \_\_m512i \_\_mm512\_mulhi\_epi16(\_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m512i \_\_mm512\_mask\_mulhi\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m512i \_\_mm512\_maskz\_mulhi\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m256i \_\_mm256\_mask\_mulhi\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHW \_\_m256i \_\_mm256\_maskz\_mulhi\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHW \_\_m128i \_\_mm\_mask\_mulhi\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULHW \_\_m128i \_\_mm\_maskz\_mulhi\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMULHW \_\_m128i \_\_mm\_mulhi\_epi16(\_\_m128i a, \_\_m128i b)  
 VPMULHW \_\_m256i \_\_mm256\_mulhi\_epi16(\_\_m256i a, \_\_m256i b)

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|-----------------------------|--------------------------|---|
| 66 0F 38 40 /r<br>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<br>VPMULLD xmm1, xmm2,<br>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<br>VPMULLD ymm1, ymm2,<br>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.                            |
| EVEX.NDS.128.66.0F38.W0 40 /r<br>VPMULLD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.W0 40 /r<br>VPMULLD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                         | AVX512VL<br>AVX512F      | Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.W0 40 /r<br>VPMULLD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                         | AVX512F                  | Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1. |
| EVEX.NDS.128.66.0F38.W1 40 /r<br>VPMULLQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                         | AVX512VL<br>AVX512DQ     | Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F38.W1 40 /r<br>VPMULLQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                         | AVX512VL<br>AVX512DQ     | Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F38.W1 40 /r<br>VPMULLQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                         | AVX512DQ                 | Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### VPMULLQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

      THEN Temp[127:0] ← SRC1[i+63:i] \* SRC2[63:0]

      ELSE Temp[127:0] ← SRC1[i+63:i] \* SRC2[i+63:i]

    FI;

    DEST[i+63:i] ← Temp[63:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VPMULLD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN Temp[63:0] ← SRC1[i+31:i] \* SRC2[31:0]

      ELSE Temp[63:0] ← SRC1[i+31:i] \* SRC2[i+31:i]

    FI;

    DEST[i+31:i] ← Temp[31:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMULLD (VEX.256 encoded version)**

Temp0[63:0] ← SRC1[31:0] \* SRC2[31:0]  
 Temp1[63:0] ← SRC1[63:32] \* SRC2[63:32]  
 Temp2[63:0] ← SRC1[95:64] \* SRC2[95:64]  
 Temp3[63:0] ← SRC1[127:96] \* SRC2[127:96]  
 Temp4[63:0] ← SRC1[159:128] \* SRC2[159:128]  
 Temp5[63:0] ← SRC1[191:160] \* SRC2[191:160]  
 Temp6[63:0] ← SRC1[223:192] \* SRC2[223:192]  
 Temp7[63:0] ← SRC1[255:224] \* SRC2[255:224]

DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[159:128] ← Temp4[31:0]  
 DEST[191:160] ← Temp5[31:0]  
 DEST[223:192] ← Temp6[31:0]  
 DEST[255:224] ← Temp7[31:0]  
 DEST[MAX\_VL-1:256] ← 0

**VPMULLD (VEX.128 encoded version)**

Temp0[63:0] ← SRC1[31:0] \* SRC2[31:0]  
 Temp1[63:0] ← SRC1[63:32] \* SRC2[63:32]  
 Temp2[63:0] ← SRC1[95:64] \* SRC2[95:64]  
 Temp3[63:0] ← SRC1[127:96] \* SRC2[127:96]  
 DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[MAX\_VL-1:128] ← 0

**PMULLD (128-bit Legacy SSE version)**

Temp0[63:0] ← DEST[31:0] \* SRC[31:0]  
 Temp1[63:0] ← DEST[63:32] \* SRC[63:32]  
 Temp2[63:0] ← DEST[95:64] \* SRC[95:64]  
 Temp3[63:0] ← DEST[127:96] \* SRC[127:96]  
 DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULLD \_\_m512i \_\_mm512\_mullo\_epi32(\_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m512i \_\_mm512\_mask\_mullo\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m512i \_\_mm512\_maskz\_mullo\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m256i \_\_mm256\_mask\_mullo\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLD \_\_m256i \_\_mm256\_maskz\_mullo\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLD \_\_m128i \_\_mm\_mask\_mullo\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLD \_\_m128i \_\_mm\_maskz\_mullo\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLD \_\_m256i \_\_mm256\_mullo\_epi32(\_\_m256i a, \_\_m256i b);  
 PMULLD \_\_m128i \_\_mm\_mullo\_epi32(\_\_m128i a, \_\_m128i b);  
 VPMULLQ \_\_m512i \_\_mm512\_mullo\_epi64(\_\_m512i a, \_\_m512i b);  
 VPMULLQ \_\_m512i \_\_mm512\_mask\_mullo\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);



VPMULLQ \_\_m512i \_\_mm512\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
VPMULLQ \_\_m256i \_\_mm256\_mullo\_epi64(\_\_m256i a, \_\_m256i b);  
VPMULLQ \_\_m256i \_\_mm256\_mask\_mullo\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPMULLQ \_\_m256i \_\_mm256\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPMULLQ \_\_m128i \_\_mm\_mullo\_epi64(\_\_m128i a, \_\_m128i b);  
VPMULLQ \_\_m128i \_\_mm\_mask\_mullo\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
VPMULLQ \_\_m128i \_\_mm\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMULLW—Multiply Packed Integers and Store Low Result

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| 66 0F D5 /r<br>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 D5 /r<br>VPMULLW xmm1, xmm2,<br>xmm3/m128              | RVM       | V/V                          | AVX                      | Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1.                    |
| VEX.NDS.256.66.0F D5 /r<br>VPMULLW ymm1, ymm2,<br>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.                    |
| EVEX.NDS.128.66.0F.WIG D5 /r<br>VPMULLW xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.WIG D5 /r<br>VPMULLW ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.WIG D5 /r<br>VPMULLW zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1. |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD signed multiply of the packed signed word (dword) integers in the first source operand and the second source operand and stores the low 16 bits of each intermediate 32-bit result 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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The first source and destination operands are ZMM/YMM/XMM registers.

### Operation

#### PMULLW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR  $j \leftarrow 0$  TO KL-1

```

i ← j * 16
IF k1[j] OR *no writemask*
  THEN
    temp[31:0] ← SRC1[j+15:i] * SRC2[j+15:i]
    DEST[j+15:i] ← temp[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[j+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMULLW (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[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[143:128] ← Temp8[15:0]
DEST[159:144] ← Temp9[15:0]
DEST[175:160] ← Temp10[15:0]
DEST[191:176] ← Temp11[15:0]
DEST[207:192] ← Temp12[15:0]
DEST[223:208] ← Temp13[15:0]
DEST[239:224] ← Temp14[15:0]
DEST[255:240] ← Temp15[15:0]
DEST[MAX_VL-1:256] ← 0

```

**VPMULLW (VEX.128 encoded version)**

```

Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]

```

Temp3[31:0] ← SRC1[63:48] \* SRC2[63:48]  
 Temp4[31:0] ← SRC1[79:64] \* SRC2[79:64]  
 Temp5[31:0] ← SRC1[95:80] \* SRC2[95:80]  
 Temp6[31:0] ← SRC1[111:96] \* SRC2[111:96]  
 Temp7[31:0] ← SRC1[127:112] \* SRC2[127:112]  
 DEST[15:0] ← Temp0[15:0]  
 DEST[31:16] ← Temp1[15:0]  
 DEST[47:32] ← Temp2[15:0]  
 DEST[63:48] ← Temp3[15:0]  
 DEST[79:64] ← Temp4[15:0]  
 DEST[95:80] ← Temp5[15:0]  
 DEST[111:96] ← Temp6[15:0]  
 DEST[127:112] ← Temp7[15:0]  
 DEST[MAX\_VL-1:128] ← 0

**PMULLW (128-bit Legacy SSE version)**

Temp0[31:0] ← DEST[15:0] \* SRC[15:0]  
 Temp1[31:0] ← DEST[31:16] \* SRC[31:16]  
 Temp2[31:0] ← DEST[47:32] \* SRC[47:32]  
 Temp3[31:0] ← DEST[63:48] \* SRC[63:48]  
 Temp4[31:0] ← DEST[79:64] \* SRC[79:64]  
 Temp5[31:0] ← DEST[95:80] \* SRC[95:80]  
 Temp6[31:0] ← DEST[111:96] \* SRC[111:96]  
 Temp7[31:0] ← DEST[127:112] \* SRC[127:112]  
 DEST[15:0] ← Temp0[15:0]  
 DEST[31:16] ← Temp1[15:0]  
 DEST[47:32] ← Temp2[15:0]  
 DEST[63:48] ← Temp3[15:0]  
 DEST[79:64] ← Temp4[15:0]  
 DEST[95:80] ← Temp5[15:0]  
 DEST[111:96] ← Temp6[15:0]  
 DEST[127:112] ← Temp7[15:0]  
 DEST[127:96] ← Temp3[31:0];  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULLW \_\_m512i \_\_mm512\_mullo\_epi16(\_\_m512i a, \_\_m512i b);  
 VPMULLW \_\_m512i \_\_mm512\_mask\_mullo\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULLW \_\_m512i \_\_mm512\_maskz\_mullo\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULLW \_\_m256i \_\_mm256\_mask\_mullo\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULLW \_\_m256i \_\_mm256\_maskz\_mullo\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULLW \_\_m128i \_\_mm\_mask\_mullo\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLW \_\_m128i \_\_mm\_maskz\_mullo\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMULLW \_\_m128i \_\_mm\_mullo\_epi16 (\_\_m128i a, \_\_m128i b);  
 VPMULLW \_\_m256i \_\_mm256\_mullo\_epi16 (\_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Sources

| Opcode / Instruction  | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag     | Description   |
|---|--------|------------------------|------------------------|---|
| EVEX.NDS.128.66.0F38.W1 83 /r<br>VPMULTISHIFTQB xmm1 {k1}{z},<br>xmm2,xmm3/m128/m64bcst | FV     | V/V                    | AVX512VBMI<br>AVX512VL | Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1. |
| EVEX.NDS.256.66.0F38.W1 83 /r<br>VPMULTISHIFTQB ymm1 {k1}{z},<br>ymm2,ymm3/m256/m64bcst | FV     | V/V                    | AVX512VBMI<br>AVX512VL | Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1. |
| EVEX.NDS.512.66.0F38.W1 83 /r<br>VPMULTISHIFTQB zmm1 {k1}{z},<br>zmm2,zmm3/m512/m64bcst | FV     | V/V                    | AVX512VBMI             | Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is left-shifted from the beginning of the input qword source by the amount specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.

### Operation

#### VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $i \leftarrow 0$  TO  $KL-1$

$q \leftarrow i * 64$ ;

  IF src2 is Memory AND EVEX.b=1 THEN

$tcur64[63:0] \leftarrow src2[63:0]$ ;

  ELSE

$tcur64[63:0] \leftarrow src2[q+63:q]$ ;

  FI;

  FOR  $j \leftarrow 0$  to 7 // iterate each byte in qword

$ctrl \leftarrow src1[q+j*8+7:q+j*8] \& 63$ ;

    FOR  $k \leftarrow 0$  to 7 // iterate each bit in byte

$tmp8[k] \leftarrow tcur64[(ctrl+k) \& 63]$ ;

    ENDFOR

    IF  $k1[i*8+j]$  or no writemask THEN

$dst[q+j*8+7:q+j*8] \leftarrow tmp8[7:0]$ ;

```

    ELSE IF zeroing-masking THEN
        dst[ q+ j*8 + 7: q + j*8] ← 0;
    ENDFOR
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VPMULTISHIFTQB __m512i _mm512_multishift_epi64_epi8( __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_maskz_multishift_epi64_epi8( __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i _mm256_multishift_epi64_epi8( __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_maskz_multishift_epi64_epi8( __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i _mm_multishift_epi64_epi8( __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_maskz_multishift_epi64_epi8( __mmask8 k, __m128i a, __m128i b);

```

#### SIMD Floating-Point Exceptions

None.

#### Other Exceptions

See Exceptions Type E4NF.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F F4 /r<br>PMULUDQ xmm1, xmm2/m128   | RM         | V/V                          | SSE4_1                   | 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<br>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<br>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.                            |
| EVEX.NDS.128.66.0F.W1 F4 /r<br>VPMULUDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1. |
| EVEX.NDS.256.66.0F.W1 F4 /r<br>VPMULUDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1. |
| EVEX.NDS.512.66.0F.W1 F4 /r<br>VPMULUDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1. |

### 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Multiplies packed unsigned doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed unsigned doubleword integers in the corresponding elements of the second source operand and stores packed unsigned quadword results in the destination operand.

128-bit Legacy SSE version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source

operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

### Operation

#### VPMULUDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i] ) \* ZeroExtend64( SRC2[31:0] )

      ELSE DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i] ) \* ZeroExtend64( SRC2[i+31:i] )

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

      ELSE \*zeroing-masking\* ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VPMULUDQ (VEX.256 encoded version)

DEST[63:0] ← ZeroExtend64( SRC1[31:0] ) \* ZeroExtend64( SRC2[31:0] )

DEST[127:64] ← ZeroExtend64( SRC1[95:64] ) \* ZeroExtend64( SRC2[95:64] )

DEST[191:128] ← ZeroExtend64( SRC1[159:128] ) \* ZeroExtend64( SRC2[159:128] )

DEST[255:192] ← ZeroExtend64( SRC1[223:192] ) \* ZeroExtend64( SRC2[223:192] )

DEST[MAX\_VL-1:256] ← 0

#### VPMULUDQ (VEX.128 encoded version)

DEST[63:0] ← ZeroExtend64( SRC1[31:0] ) \* ZeroExtend64( SRC2[31:0] )

DEST[127:64] ← ZeroExtend64( SRC1[95:64] ) \* ZeroExtend64( SRC2[95:64] )

DEST[MAX\_VL-1:128] ← 0

#### PMULUDQ (128-bit Legacy SSE version)

DEST[63:0] ← DEST[31:0] \* SRC[31:0]

DEST[127:64] ← DEST[95:64] \* SRC[95:64]

DEST[MAX\_VL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

VPMULUDQ \_\_m512i \_\_mm512\_mul\_epu32(\_\_m512i a, \_\_m512i b);

VPMULUDQ \_\_m512i \_\_mm512\_mask\_mul\_epu32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULUDQ \_\_m512i \_\_mm512\_maskz\_mul\_epu32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULUDQ \_\_m256i \_\_mm256\_mask\_mul\_epu32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULUDQ \_\_m256i \_\_mm256\_maskz\_mul\_epu32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULUDQ \_\_m128i \_\_mm\_mask\_mul\_epu32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULUDQ \_\_m128i \_\_mm\_maskz\_mul\_epu32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMULUDQ \_\_m128i \_\_mm\_mul\_epu32(\_\_m128i a, \_\_m128i b);



VPMULUDQ \_\_m256i \_mm256\_mul\_epu32( \_\_m256i a, \_\_m256i b);

#### **SIMD Floating-Point Exceptions**

None

#### **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## POR—Bitwise Logical Or

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| 66 0F EB /r<br>POR xmm1, xmm2/m128  | RM        | V/V                          | SSE2                     | Bitwise OR of xmm2/m128 and xmm1.  |
| VEX.NDS.128.66.0F.WIG EB /r<br>VPOR xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX                      | Bitwise OR of xmm2/m128 and xmm3.  |
| VEX.NDS.256.66.0F.WIG EB /r<br>VPOR ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Bitwise OR of ymm2/m256 and ymm3.  |
| EVEX.NDS.128.66.0F.W0 EB /r<br>VPORD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1. |
| EVEX.NDS.256.66.0F.W0 EB /r<br>VPORD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1. |
| EVEX.NDS.512.66.0F.W0 EB /r<br>VPORD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1. |
| EVEX.NDS.128.66.0F.W1 EB /r<br>VPORQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.   |
| EVEX.NDS.256.66.0F.W1 EB /r<br>VPORQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.   |
| EVEX.NDS.512.66.0F.W1 EB /r<br>VPORQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.   |

### 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a bitwise logical OR operation on the second source operand and the first source operand and stores the result in the destination operand. Each bit of the result is set to 0 if the corresponding bits of the first and second operands are 0; otherwise, it is set to 1.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

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 (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

## Operation

### VPORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]

      ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

### VPORQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[63:0]

      ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

### VPOR (VEX.256 encoded version)

DEST[255:0] ← SRC1 OR SRC2

DEST[MAX\_VL-1:256] ← 0

### VPOR (VEX.128 encoded version)

DEST[127:0] ← (SRC[127:0] OR SRC2[127:0])  
 DEST[MAX\_VL-1:128] ← 0

**POR (128-bit Legacy SSE version)**

DEST[127:0] ← (SRC[127:0] OR SRC2[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPORD \_\_m512i \_mm512\_or\_epi32(\_\_m512i a, \_\_m512i b);  
 VPORD \_\_m512i \_mm512\_mask\_or\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPORD \_\_m512i \_mm512\_maskz\_or\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPORD \_\_m256i \_mm256\_or\_epi32(\_\_m256i a, \_\_m256i b);  
 VPORD \_\_m256i \_mm256\_mask\_or\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORD \_\_m256i \_mm256\_maskz\_or\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORD \_\_m128i \_mm\_or\_epi32(\_\_m128i a, \_\_m128i b);  
 VPORD \_\_m128i \_mm\_mask\_or\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORD \_\_m128i \_mm\_maskz\_or\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m512i \_mm512\_or\_epi64(\_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m512i \_mm512\_mask\_or\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m512i \_mm512\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m256i \_mm256\_or\_epi64(\_\_m256i a, int imm);  
 VPORQ \_\_m256i \_mm256\_mask\_or\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORQ \_\_m256i \_mm256\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORQ \_\_m128i \_mm\_or\_epi64(\_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m128i \_mm\_mask\_or\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m128i \_mm\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)POR \_\_m128i \_mm\_or\_si128 (\_\_m128i a, \_\_m128i b)  
 VPOR \_\_m256i \_mm256\_or\_si256 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

none

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PROLD/PROLVD/PROLQ/PROLVQ—Bit Rotate Left

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F38.W0 15 /r<br>VPROLVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.    |
| EVEX.NDD.128.66.0F.W0 72 /1 ib<br>VPROLD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.  |
| EVEX.NDS.128.66.0F38.W1 15 /r<br>VPROLVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.      |
| EVEX.NDD.128.66.0F.W1 72 /1 ib<br>VPROLQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.  |
| EVEX.NDS.256.66.0F38.W0 15 /r<br>VPROLVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.    |
| EVEX.NDD.256.66.0F.W0 72 /1 ib<br>VPROLD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.  |
| EVEX.NDS.256.66.0F38.W1 15 /r<br>VPROLVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.      |
| EVEX.NDD.256.66.0F.W1 72 /1 ib<br>VPROLQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.  |
| EVEX.NDS.512.66.0F38.W0 15 /r<br>VPROLVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV-RVM  | V/V                          | AVX512F                  | Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1. |
| EVEX.NDD.512.66.0F.W0 72 /1 ib<br>VPROLD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | FV-VMI  | V/V                          | AVX512F                  | Rotate left of doublewords in zmm2/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.                                       |
| EVEX.NDS.512.66.0F38.W1 15 /r<br>VPROLVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV-RVM  | V/V                          | AVX512F                  | Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.      |
| EVEX.NDD.512.66.0F.W1 72 /1 ib<br>VPROLQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | FV-VMI  | V/V                          | AVX512F                  | Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.  |

## Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|--------|---------------|---------------|---------------|-----------|
| FV-VMI | VEX.vvvv (w)  | ModRM:r/m (R) | Imm8          | NA        |
| FV-RVM | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

## Operation

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC modulo 32;
```

```
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));
```

```
LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC modulo 64;
```

```
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));
```

## VPROLD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
```

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 32
```

```
  IF k1[j] OR *no writemask* THEN
```

```
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
```

```
      THEN DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
```

```
      ELSE DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
```

```
    FI;
```

```
  ELSE
```

```
    IF *merging-masking* ;merging-masking
```

```
      THEN *DEST[i+31:i] remains unchanged*
```

```
      ELSE *zeroing-masking* ;zeroing-masking
```

```
        DEST[i+31:i] ← 0
```

```
    FI
```

```
  FI;
```

```
ENDFOR
```

```
DEST[MAX_VL-1:VL] ← 0
```

**VPROLVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[31:0])

ELSE DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPROLQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPROLVQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPROLD \_\_m512i \_mm512\_rol\_epi32(\_\_m512i a, int imm);  
 VPROLD \_\_m512i \_mm512\_mask\_rol\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, int imm);  
 VPROLD \_\_m512i \_mm512\_maskz\_rol\_epi32(\_\_mmask16 k, \_\_m512i a, int imm);  
 VPROLD \_\_m256i \_mm256\_rol\_epi32(\_\_m256i a, int imm);  
 VPROLD \_\_m256i \_mm256\_mask\_rol\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPROLD \_\_m256i \_mm256\_maskz\_rol\_epi32(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPROLD \_\_m128i \_mm\_rol\_epi32(\_\_m128i a, int imm);  
 VPROLD \_\_m128i \_mm\_mask\_rol\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPROLD \_\_m128i \_mm\_maskz\_rol\_epi32(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPROLQ \_\_m512i \_mm512\_rol\_epi64(\_\_m512i a, int imm);  
 VPROLQ \_\_m512i \_mm512\_mask\_rol\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, int imm);  
 VPROLQ \_\_m512i \_mm512\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m512i a, int imm);  
 VPROLQ \_\_m256i \_mm256\_rol\_epi64(\_\_m256i a, int imm);  
 VPROLQ \_\_m256i \_mm256\_mask\_rol\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPROLQ \_\_m256i \_mm256\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPROLQ \_\_m128i \_mm\_rol\_epi64(\_\_m128i a, int imm);  
 VPROLQ \_\_m128i \_mm\_mask\_rol\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPROLQ \_\_m128i \_mm\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPROLVD \_\_m512i \_mm512\_rolv\_epi32(\_\_m512i a, \_\_m512i cnt);  
 VPROLVD \_\_m512i \_mm512\_mask\_rolv\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, \_\_m512i cnt);  
 VPROLVD \_\_m512i \_mm512\_maskz\_rolv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPROLVD \_\_m256i \_mm256\_rolv\_epi32(\_\_m256i a, \_\_m256i cnt);  
 VPROLVD \_\_m256i \_mm256\_mask\_rolv\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPROLVD \_\_m256i \_mm256\_maskz\_rolv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPROLVD \_\_m128i \_mm\_rolv\_epi32(\_\_m128i a, \_\_m128i cnt);  
 VPROLVD \_\_m128i \_mm\_mask\_rolv\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPROLVD \_\_m128i \_mm\_maskz\_rolv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPROLVQ \_\_m512i \_mm512\_rolv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPROLVQ \_\_m512i \_mm512\_mask\_rolv\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, \_\_m512i cnt);  
 VPROLVQ \_\_m512i \_mm512\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPROLVQ \_\_m256i \_mm256\_rolv\_epi64(\_\_m256i a, \_\_m256i cnt);  
 VPROLVQ \_\_m256i \_mm256\_mask\_rolv\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPROLVQ \_\_m256i \_mm256\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPROLVQ \_\_m128i \_mm\_rolv\_epi64(\_\_m128i a, \_\_m128i cnt);  
 VPROLVQ \_\_m128i \_mm\_mask\_rolv\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPROLVQ \_\_m128i \_mm\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.



**PRORD/PRORVD/PRORQ/PRORVQ—Bit Rotate Right**

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| EEX.NDS.128.66.0F38.W0 14 /r<br>VPRORVD xmm1 {k1}{z},<br>xmm2,<br>xmm3/m128/m32bcst | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1. |
| EEX.NDD.128.66.0F.W0 72 /0 ib<br>VPRORD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8    | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.                                       |
| EEX.NDS.128.66.0F38.W1 14 /r<br>VPRORVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst    | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.   |
| EEX.NDD.128.66.0F.W1 72 /0 ib<br>VPRORQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8    | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.   |
| EEX.NDS.256.66.0F38.W0 14 /r<br>VPRORVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst    | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1. |
| EEX.NDD.256.66.0F.W0 72 /0 ib<br>VPRORD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8    | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.                                       |
| EEX.NDS.256.66.0F38.W1 14 /r<br>VPRORVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst    | FV-RVM  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.   |
| EEX.NDD.256.66.0F.W1 72 /0 ib<br>VPRORQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8    | FV-VMI  | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.   |
| EEX.NDS.512.66.0F38.W0 14 /r<br>VPRORVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst    | FV-RVM  | V/V                          | AVX512F                  | Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1. |
| EEX.NDD.512.66.0F.W0 72 /0 ib<br>VPRORD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8    | FV-VMI  | V/V                          | AVX512F                  | Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.                                       |
| EEX.NDS.512.66.0F38.W1 14 /r<br>VPRORVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst    | FV-RVM  | V/V                          | AVX512F                  | Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.   |
| EEX.NDD.512.66.0F.W1 72 /0 ib<br>VPRORQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8    | FV-VMI  | V/V                          | AVX512F                  | Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.   |

## Instruction Operand Encoding

| Op/En  | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|--------|---------------|---------------|---------------|-----------|
| FV-VMI | VEX.vvvv (w)  | ModRM:r/m (R) | Imm8          | NA        |
| FV-RVM | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

## Operation

RIGHT\_ROTATE\_DWORDS(SRC, COUNT\_SRC)

COUNT ← COUNT\_SRC modulo 32;

DEST[31:0] ← (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT\_ROTATE\_QWORDS(SRC, COUNT\_SRC)

COUNT ← COUNT\_SRC modulo 64;

DEST[63:0] ← (SRC >> COUNT) | (SRC << (64 - COUNT));

## VPRORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

      THEN DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS( SRC1[31:0], imm8)

      ELSE DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], imm8)

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPRORVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[31:0])

ELSE DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPRORQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPRORVQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPRORD \_\_m512i \_mm512\_ror\_epi32(\_\_m512i a, int imm);  
 VPRORD \_\_m512i \_mm512\_mask\_ror\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, int imm);  
 VPRORD \_\_m512i \_mm512\_maskz\_ror\_epi32(\_\_mmask16 k, \_\_m512i a, int imm);  
 VPRORD \_\_m256i \_mm256\_ror\_epi32(\_\_m256i a, int imm);  
 VPRORD \_\_m256i \_mm256\_mask\_ror\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPRORD \_\_m256i \_mm256\_maskz\_ror\_epi32(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPRORD \_\_m128i \_mm\_ror\_epi32(\_\_m128i a, int imm);  
 VPRORD \_\_m128i \_mm\_mask\_ror\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPRORD \_\_m128i \_mm\_maskz\_ror\_epi32(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPRORQ \_\_m512i \_mm512\_ror\_epi64(\_\_m512i a, int imm);  
 VPRORQ \_\_m512i \_mm512\_mask\_ror\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, int imm);  
 VPRORQ \_\_m512i \_mm512\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m512i a, int imm);  
 VPRORQ \_\_m256i \_mm256\_ror\_epi64(\_\_m256i a, int imm);  
 VPRORQ \_\_m256i \_mm256\_mask\_ror\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPRORQ \_\_m256i \_mm256\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPRORQ \_\_m128i \_mm\_ror\_epi64(\_\_m128i a, int imm);  
 VPRORQ \_\_m128i \_mm\_mask\_ror\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPRORQ \_\_m128i \_mm\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPRORVD \_\_m512i \_mm512\_rorv\_epi32(\_\_m512i a, \_\_m512i cnt);  
 VPRORVD \_\_m512i \_mm512\_mask\_rorv\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, \_\_m512i cnt);  
 VPRORVD \_\_m512i \_mm512\_maskz\_rorv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPRORVD \_\_m256i \_mm256\_rorv\_epi32(\_\_m256i a, \_\_m256i cnt);  
 VPRORVD \_\_m256i \_mm256\_mask\_rorv\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPRORVD \_\_m256i \_mm256\_maskz\_rorv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPRORVD \_\_m128i \_mm\_rorv\_epi32(\_\_m128i a, \_\_m128i cnt);  
 VPRORVD \_\_m128i \_mm\_mask\_rorv\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPRORVD \_\_m128i \_mm\_maskz\_rorv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPRORVQ \_\_m512i \_mm512\_rorv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPRORVQ \_\_m512i \_mm512\_mask\_rorv\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, \_\_m512i cnt);  
 VPRORVQ \_\_m512i \_mm512\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPRORVQ \_\_m256i \_mm256\_rorv\_epi64(\_\_m256i a, \_\_m256i cnt);  
 VPRORVQ \_\_m256i \_mm256\_mask\_rorv\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPRORVQ \_\_m256i \_mm256\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPRORVQ \_\_m128i \_mm\_rorv\_epi64(\_\_m128i a, \_\_m128i cnt);  
 VPRORVQ \_\_m128i \_mm\_mask\_rorv\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPRORVQ \_\_m128i \_mm\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 A0 /vsib<br>VPSCATTERDD vm32x {k1}, xmm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter dword values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A0 /vsib<br>VPSCATTERDD vm32y {k1}, ymm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter dword values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A0 /vsib<br>VPSCATTERDD vm32z {k1}, zmm1 | T1S       | V/V                         | AVX512F                  | Using signed dword indices, scatter dword values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A0 /vsib<br>VPSCATTERDQ vm32x {k1}, xmm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter qword values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A0 /vsib<br>VPSCATTERDQ vm32x {k1}, ymm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter qword values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A0 /vsib<br>VPSCATTERDQ vm32y {k1}, zmm1 | T1S       | V/V                         | AVX512F                  | Using signed dword indices, scatter qword values to memory using writemask k1. |
| EVEX.128.66.0F38.W0 A1 /vsib<br>VPSCATTERQD vm64x {k1}, xmm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter dword values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A1 /vsib<br>VPSCATTERQD vm64y {k1}, xmm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter dword values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A1 /vsib<br>VPSCATTERQD vm64z {k1}, ymm1 | T1S       | V/V                         | AVX512F                  | Using signed qword indices, scatter dword values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A1 /vsib<br>VPSCATTERQQ vm64x {k1}, xmm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter qword values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A1 /vsib<br>VPSCATTERQQ vm64y {k1}, ymm1 | T1S       | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter qword values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A1 /vsib<br>VPSCATTERQQ vm64z {k1}, zmm1 | T1S       | V/V                         | AVX512F                  | Using signed qword indices, scatter qword values to memory using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1   | Operand 2     | Operand 3 | Operand 4 |
|-------|---|---------------|-----------|-----------|
| T1S   | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | ModRM:reg (r) | NA        | NA        |

### Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the `VSIB` (i.e., the index register is a vector register, holding packed indices). Elements

will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (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 register are partially updated. If any traps or interrupts are pending from already scattered 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.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- 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 ZMM 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 scattered 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.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

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.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VPSCATTERDD (EVEX encoded versions)**

(KL, VL)= (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + SignExtend(VINDEX[i+31:i]) \* SCALE + DISP] ← SRC[i+31:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERDQ (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP] ← SRC[i+63:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERQD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEX[k+63:k]) \* SCALE + DISP] ← SRC[i+31:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERQQ (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEX[j+63:j]) \* SCALE + DISP] ← SRC[i+63:i]

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSCATTERDD void \_\_mm512\_i32scatter\_epi32(void \* base, \_\_m512i vdx, \_\_m512i a, int scale);  
 VPSCATTERDD void \_\_mm256\_i32scatter\_epi32(void \* base, \_\_m256i vdx, \_\_m256i a, int scale);  
 VPSCATTERDD void \_\_mm\_i32scatter\_epi32(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERDD void \_\_mm512\_mask\_i32scatter\_epi32(void \* base, \_\_mmask16 k, \_\_m512i vdx, \_\_m512i a, int scale);  
 VPSCATTERDD void \_\_mm256\_mask\_i32scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256i a, int scale);  
 VPSCATTERDD void \_\_mm\_mask\_i32scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERDQ void \_\_mm512\_i32scatter\_epi64(void \* base, \_\_m256i vdx, \_\_m512i a, int scale);  
 VPSCATTERDQ void \_\_mm256\_i32scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m256i a, int scale);  
 VPSCATTERDQ void \_\_mm\_i32scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERDQ void \_\_mm512\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m512i a, int scale);  
 VPSCATTERDQ void \_\_mm256\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m256i a, int scale);  
 VPSCATTERDQ void \_\_mm\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERQD void \_\_mm512\_i64scatter\_epi32(void \* base, \_\_m512i vdx, \_\_m256i a, int scale);  
 VPSCATTERQD void \_\_mm256\_i64scatter\_epi32(void \* base, \_\_m256i vdx, \_\_m128i a, int scale);  
 VPSCATTERQD void \_\_mm\_i64scatter\_epi32(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERQD void \_\_mm512\_mask\_i64scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m256i a, int scale);  
 VPSCATTERQD void \_\_mm256\_mask\_i64scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m128i a, int scale);  
 VPSCATTERQD void \_\_mm\_mask\_i64scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERQQ void \_\_mm512\_i64scatter\_epi64(void \* base, \_\_m512i vdx, \_\_m512i a, int scale);  
 VPSCATTERQQ void \_\_mm256\_i64scatter\_epi64(void \* base, \_\_m256i vdx, \_\_m256i a, int scale);  
 VPSCATTERQQ void \_\_mm\_i64scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);  
 VPSCATTERQQ void \_\_mm512\_mask\_i64scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m512i a, int scale);  
 VPSCATTERQQ void \_\_mm256\_mask\_i64scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256i a, int scale);  
 VPSCATTERQQ void \_\_mm\_mask\_i64scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.



## PSHUFB—Packed Shuffle Bytes

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|-----------------------------|--------------------------|---|
| 66 0F 38 00 /r<br>PSHUFB xmm1, xmm2/m128                                   | RM         | V/V                         | SSSE3                    | Shuffle bytes in xmm1 according to contents of xmm2/m128.                     |
| VEX.NDS.128.66.0F38 00 /r<br>VPSHUFB xmm1, xmm2, xmm3/m128                 | RVM        | V/V                         | AVX                      | Shuffle bytes in xmm2 according to contents of xmm3/m128.                     |
| VEX.NDS.256.66.0F38 00 /r<br>VPSHUFB ymm1, ymm2, ymm3/m256                 | RVM        | V/V                         | AVX2                     | Shuffle bytes in ymm2 according to contents of ymm3/m256.                     |
| EVEX.NDS.128.66.0F38.WIG 00 /r<br>VPSHUFB xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1. |
| EVEX.NDS.256.66.0F38.WIG 00 /r<br>VPSHUFB ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                         | AVX512VL<br>AVX512BW     | Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1. |
| EVEX.NDS.512.66.0F38.WIG 00 /r<br>VPSHUFB zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                         | AVX512BW                 | Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA            | NA        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

PSHUFB performs in-place shuffles of bytes in the first source operand according to the shuffle control mask in the second source operand. The instruction permutes the data in the first source 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 first source operand. The value of each index is the least significant 4 bits of the shuffle control byte. The first source and destination operands are vector registers. The second source is either a vector register or a memory location.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:128) of the destination 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.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four in-lane 128-bit shuffles.

**Operation****VPSHUFB (EVEX.512 encoded version)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN

IF (SRC2[(i \* 8)+7] = 1 ) then

THEN DEST[i+7:i] ← 0;

ELSE

index[3:0] ← SRC2[i+3:i];

OFFSET = j MODULO 128

DEST[i+7:i] ← SRC1[OFFSET + index\*8+7:OFFSET + index\*8];

FI;

ELSE

IF \*merging-masking\*                   ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\*               ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 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

}

DEST[MAX\_VL-1:256] ← 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[MAX\_VL-1:128] ← 0

**PSHUFB (128-bit Legacy SSE version)**

```

for i = 0 to 15 {
  if (SRC[(i * 8)+7] = 1 ) then
    DEST[(i*8)+7:(i*8)+0] ← 0;
  else
    index[3:0] ← SRC[(i*8)+3 : (i*8)+0];
    DEST[(i*8)+7:(i*8)+0] ← DEST[(index*8+7):(index*8+0)];
  endif
}
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
(V)PSHUFB __m128i _mm_shuffle_epi8(__m128i a, __m128i b)
VPSHUFB __m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSHUFHW—Shuffle Packed High Words

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| F3 0F 70 /r ib<br>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 70 /r ib<br>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 70 /r ib<br>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.                     |
| EVEX.128.F3.0F.WIG 70 /r ib<br>VPSHUFHW xmm1 {k1}{z},<br>xmm2/m128, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1. |
| EVEX.256.F3.0F.WIG 70 /r ib<br>VPSHUFHW ymm1 {k1}{z},<br>ymm2/m256, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1. |
| EVEX.512.F3.0F.WIG 70 /r ib<br>VPSHUFHW zmm1 {k1}{z},<br>zmm2/m512, imm8 | FVM        | V/V                          | AVX512BW                 | Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RMI   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |
| FVM   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Copies words from the high quadword of the source operand and inserts them in the high quadword of the destination operand at word locations selected with the immediate operand. This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure F-1. For the PSHUFHW instruction, 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) 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.

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.

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: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAX\_VL-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 (MAX\_VL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

## Operation

### VPSHUFHW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

```
TMP_DEST[63:0] ← SRC1[63:0]
TMP_DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
TMP_DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
TMP_DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
TMP_DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
```

FI;

IF VL >= 256

```
TMP_DEST[191:128] ← SRC1[191:128]
TMP_DEST[207:192] ← (SRC1 >> (imm[1:0] * 16))[207:192]
TMP_DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
TMP_DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
TMP_DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
```

FI;

IF VL >= 512

```
TMP_DEST[319:256] ← SRC1[319:256]
TMP_DEST[335:320] ← (SRC1 >> (imm[1:0] * 16))[335:320]
TMP_DEST[351:336] ← (SRC1 >> (imm[3:2] * 16))[335:320]
TMP_DEST[367:352] ← (SRC1 >> (imm[5:4] * 16))[335:320]
TMP_DEST[383:368] ← (SRC1 >> (imm[7:6] * 16))[335:320]
TMP_DEST[447:384] ← SRC1[447:384]
TMP_DEST[463:448] ← (SRC1 >> (imm[1:0] * 16))[463:448]
TMP_DEST[479:464] ← (SRC1 >> (imm[3:2] * 16))[463:448]
TMP_DEST[495:480] ← (SRC1 >> (imm[5:4] * 16))[463:448]
TMP_DEST[511:496] ← (SRC1 >> (imm[7:6] * 16))[463:448]
```

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i];

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 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[207:192] ← (SRC1 >> (imm[1:0] \* 16))[207:192]  
 DEST[223:208] ← (SRC1 >> (imm[3:2] \* 16))[207:192]  
 DEST[239:224] ← (SRC1 >> (imm[5:4] \* 16))[207:192]  
 DEST[255:240] ← (SRC1 >> (imm[7:6] \* 16))[207:192]  
 DEST[MAX\_VL-1:256] ← 0 ;

**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[MAX\_VL-1:128] ← 0

**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[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSHUFHW \_\_m512i \_\_mm512\_shufflehi\_epi16(\_\_m512i a, int n);  
 VPSHUFHW \_\_m512i \_\_mm512\_mask\_shufflehi\_epi16(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFHW \_\_m512i \_\_mm512\_maskz\_shufflehi\_epi16(\_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFHW \_\_m256i \_\_mm256\_mask\_shufflehi\_epi16(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFHW \_\_m256i \_\_mm256\_maskz\_shufflehi\_epi16(\_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFHW \_\_m128i \_\_mm\_mask\_shufflehi\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, int n);  
 VPSHUFHW \_\_m128i \_\_mm\_maskz\_shufflehi\_epi16(\_\_mmask8 k, \_\_m128i a, int n);  
 (V)PSHUFHW \_\_m128i \_\_mm\_shufflehi\_epi16(\_\_m128i a, int n)  
 VPSHUFHW \_\_m256i \_\_mm256\_shufflehi\_epi16(\_\_m256i a, int n)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PSHUFLW—Shuffle Packed Low Words

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| F2 0F 70 /r ib<br>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 70 /r ib<br>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 70 /r ib<br>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.                     |
| EVEX.128.F2.0F.WIG 70 /r ib<br>VPSHUFLW xmm1 {k1}{z},<br>xmm2/m128, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1. |
| EVEX.256.F2.0F.WIG 70 /r ib<br>VPSHUFLW ymm1 {k1}{z},<br>ymm2/m256, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1. |
| EVEX.512.F2.0F.WIG 70 /r ib<br>VPSHUFLW zmm1 {k1}{z},<br>zmm2/m512, imm8 | FVM       | V/V                          | AVX512BW                 | Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RMI   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |
| FVM   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Copies words from the low quadword of the source operand and inserts them in the low quadword of the destination operand at word locations selected with the immediate operand. This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure F-1. For the PSHUFLW instruction, 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.

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.

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: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding 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 (MAX\_VL-1:128) of the corresponding register destination are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VPSHUFLW (EVEX.U1.512 encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

```
TMP_DEST[15:0] ← (SRC1 >> (imm[1:0] * 16))[15:0]
TMP_DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
TMP_DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
TMP_DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
TMP_DEST[127:64] ← SRC1[127:64]
```

FI;

IF VL >= 256

```
TMP_DEST[143:128] ← (SRC1 >> (imm[1:0] * 16))[143:128]
TMP_DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
TMP_DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
TMP_DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
TMP_DEST[255:192] ← SRC1[255:192]
```

FI;

IF VL >= 512

```
TMP_DEST[271:256] ← (SRC1 >> (imm[1:0] * 16))[271:256]
TMP_DEST[287:272] ← (SRC1 >> (imm[3:2] * 16))[271:256]
TMP_DEST[303:288] ← (SRC1 >> (imm[5:4] * 16))[271:256]
TMP_DEST[319:304] ← (SRC1 >> (imm[7:6] * 16))[271:256]
TMP_DEST[383:320] ← SRC1[383:320]
TMP_DEST[399:384] ← (SRC1 >> (imm[1:0] * 16))[399:384]
TMP_DEST[415:400] ← (SRC1 >> (imm[3:2] * 16))[399:384]
TMP_DEST[431:416] ← (SRC1 >> (imm[5:4] * 16))[399:384]
TMP_DEST[447:432] ← (SRC1 >> (imm[7:6] * 16))[399:384]
TMP_DEST[511:448] ← SRC1[511:448]
```

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i];

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0



**VPSHUFLW (VEX.256 encoded version)**

DEST[15:0] ← (SRC1 >> (imm[1:0] \* 16))[15:0]  
 DEST[31:16] ← (SRC1 >> (imm[3:2] \* 16))[15:0]  
 DEST[47:32] ← (SRC1 >> (imm[5:4] \* 16))[15:0]  
 DEST[63:48] ← (SRC1 >> (imm[7:6] \* 16))[15:0]  
 DEST[127:64] ← SRC1[127:64]  
 DEST[143:128] ← (SRC1 >> (imm[1:0] \* 16))[143:128]  
 DEST[159:144] ← (SRC1 >> (imm[3:2] \* 16))[143:128]  
 DEST[175:160] ← (SRC1 >> (imm[5:4] \* 16))[143:128]  
 DEST[191:176] ← (SRC1 >> (imm[7:6] \* 16))[143:128]  
 DEST[255:192] ← SRC1[255:192]  
 DEST[MAX\_VL-1:256] ← 0 ;

**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] ← SRC1[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**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[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSHUFLW \_\_m512i \_mm512\_shuffle\_epi16(\_\_m512i a, int n);  
 VPSHUFLW \_\_m512i \_mm512\_mask\_shuffle\_epi16(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFLW \_\_m512i \_mm512\_maskz\_shuffle\_epi16(\_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFLW \_\_m256i \_mm256\_mask\_shuffle\_epi16(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFLW \_\_m256i \_mm256\_maskz\_shuffle\_epi16(\_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFLW \_\_m128i \_mm\_mask\_shuffle\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, int n);  
 VPSHUFLW \_\_m128i \_mm\_maskz\_shuffle\_epi16(\_\_mmask8 k, \_\_m128i a, int n);  
 (V)PSHUFLW \_\_m128i \_mm\_shuffle\_epi16(\_\_m128i a, int n)  
 VPSHUFLW \_\_m256i \_mm\_shuffle\_epi16(\_\_m256i a, int n)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PSHUFD—Shuffle Packed Doublewords

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F 70 /r ib<br>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<br>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<br>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.                            |
| EVEX.128.66.0F.W0 70 /r ib<br>VPSHUFD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F.W0 70 /r ib<br>VPSHUFD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F.W0 70 /r ib<br>VPSHUFD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | FV        | V/V                          | AVX512F                  | Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RMI   | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |
| FV    | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Copies doublewords from the source operand (the second operand) and inserts them in the destination operand (the first operand) at the locations selected with the immediate operand (third operand). Figure 5-34 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand imm8. 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 encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 5-34) determines which doubleword (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. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

128-bit Legacy SSE version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

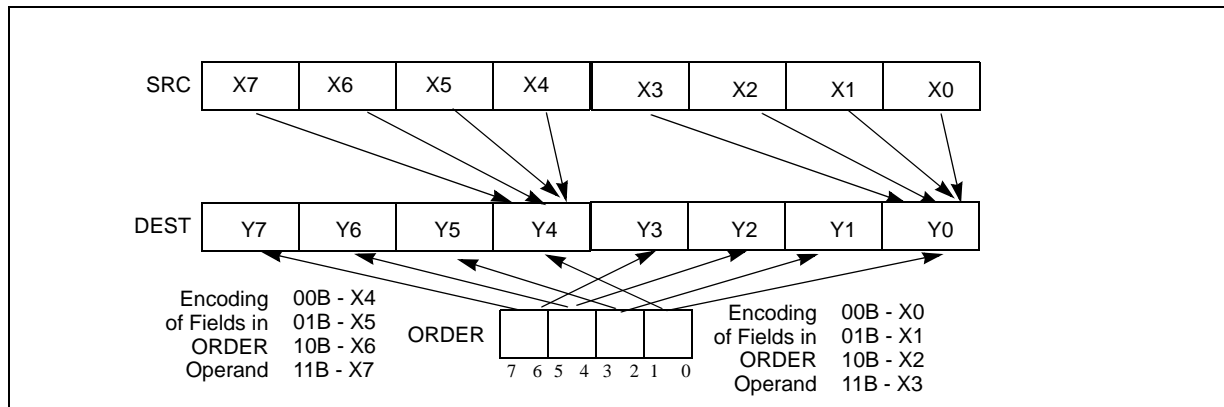


Figure 5-34. 256-bit VPSHUFD Instruction Operation

## Operation

### VPSHUFD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF (EVEX.b = 1) AND (SRC \*is memory\*)  
    THEN TMP\_SRC[i+31:i] ← SRC[31:0]  
    ELSE TMP\_SRC[i+31:i] ← SRC[i+31:i]

  FI;

ENDFOR;

IF VL ≥ 128

  TMP\_DEST[31:0] ← (TMP\_SRC[127:0] >> (ORDER[1:0] \* 32))[31:0];  
  TMP\_DEST[63:32] ← (TMP\_SRC[127:0] >> (ORDER[3:2] \* 32))[31:0];  
  TMP\_DEST[95:64] ← (TMP\_SRC[127:0] >> (ORDER[5:4] \* 32))[31:0];  
  TMP\_DEST[127:96] ← (TMP\_SRC[127:0] >> (ORDER[7:6] \* 32))[31:0];

FI;

IF VL ≥ 256

  TMP\_DEST[159:128] ← (TMP\_SRC[255:128] >> (ORDER[1:0] \* 32))[31:0];  
  TMP\_DEST[191:160] ← (TMP\_SRC[255:128] >> (ORDER[3:2] \* 32))[31:0];  
  TMP\_DEST[223:192] ← (TMP\_SRC[255:128] >> (ORDER[5:4] \* 32))[31:0];  
  TMP\_DEST[255:224] ← (TMP\_SRC[255:128] >> (ORDER[7:6] \* 32))[31:0];

FI;

IF VL ≥ 512

  TMP\_DEST[287:256] ← (TMP\_SRC[383:256] >> (ORDER[1:0] \* 32))[31:0];  
  TMP\_DEST[319:288] ← (TMP\_SRC[383:256] >> (ORDER[3:2] \* 32))[31:0];  
  TMP\_DEST[351:320] ← (TMP\_SRC[383:256] >> (ORDER[5:4] \* 32))[31:0];  
  TMP\_DEST[383:352] ← (TMP\_SRC[383:256] >> (ORDER[7:6] \* 32))[31:0];  
  TMP\_DEST[415:384] ← (TMP\_SRC[511:384] >> (ORDER[1:0] \* 32))[31:0];  
  TMP\_DEST[447:416] ← (TMP\_SRC[511:384] >> (ORDER[3:2] \* 32))[31:0];  
  TMP\_DEST[479:448] ← (TMP\_SRC[511:384] >> (ORDER[5:4] \* 32))[31:0];

```

    TMP_DEST[511:480] ← (TMP_SRC[511:384] >> (ORDER[7:6] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSHUFD (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAX_VL-1:256] ← 0

```

**VPSHUFD (VEX.128 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[MAX_VL-1:128] ← 0

```

**PSHUFD (128-bit Legacy SSE 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[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFD __m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, int n);
(V)PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n);
VPSHUFD __m256i _mm256_shuffle_epi32(__m256i a, int n);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

**PSLLDQ—Byte Shift Left**

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F 73 /7 ib<br>PSLLDQ xmm1, imm8                          | MI        | V/V                          | SSE2                     | Shift xmm1 left by imm8 bytes while shifting in 0s and store result in xmm1.      |
| VEX.NDD.128.66.0F 73 /7 ib<br>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 73 /7 ib<br>VPSLLDQ ymm1, ymm2, imm8       | VMI       | V/V                          | AVX2                     | Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.      |
| EVEX.NDD.128.66.0F 73 /7 ib<br>VPSLLDQ xmm1,xmm2/ m128, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1. |
| EVEX.NDD.256.66.0F 73 /7 ib<br>VPSLLDQ ymm1, ymm2/m256, imm8 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1. |
| EVEX.NDD.512.66.0F 73 /7 ib<br>VPSLLDQ zmm1, zmm2/m512, imm8 | FVM       | V/V                          | AVX512BW                 | Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1. |

**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        |
| FVM   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8      | NA        |

**Description**

Shifts the source operand to the left by the number of bytes specified in the count 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 source and destination operands are XMM registers. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register.

EVEX encoded versions: The source operand is a vector register or a memory location. The destination operand is a vector register.

Note: In VEX encoded versions, VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register.

**Operation****VPSLLDQ (EVEX.U1.512 encoded version)**

TEMP ← COUNT  
 IF (TEMP > 15) THEN TEMP ← 16; FI  
 DEST[127:0] ← SRC[127:0] << (TEMP \* 8)  
 DEST[255:128] ← SRC[255:128] << (TEMP \* 8)  
 DEST[383:256] ← SRC[383:256] << (TEMP \* 8)  
 DEST[511:384] ← SRC[511:384] << (TEMP \* 8)  
 DEST[MAX\_VL-1:512] ← 0

**VPSLLDQ (VEX.256 and EVEX.U1.256 encoded version)**

TEMP ← COUNT  
 IF (TEMP > 15) THEN TEMP ← 16; FI  
 DEST[127:0] ← SRC[127:0] << (TEMP \* 8)  
 DEST[255:128] ← SRC[255:128] << (TEMP \* 8)  
 DEST[MAX\_VL-1:256] ← 0

**VPSLLDQ (VEX.128 and EVEX.U1.128 encoded version)**

TEMP ← COUNT  
 IF (TEMP > 15) THEN TEMP ← 16; FI  
 DEST ← SRC << (TEMP \* 8)  
 DEST[MAX\_VL-1:128] ← 0

**PSLLDQ(128-bit Legacy SSE version)**

TEMP ← COUNT  
 IF (TEMP > 15) THEN TEMP ← 16; FI  
 DEST ← DEST << (TEMP \* 8)  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PSLLDQ \_\_m128i \_mm\_bslli\_epi128 ( \_\_m128i a, int imm)  
 VPSLLDQ \_\_m256i \_mm256\_bslli\_epi128 ( \_\_m256i a, const int imm)  
 VPSLLDQ \_\_m512i \_mm512\_bslli\_epi128 ( \_\_m512i a, const int imm)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.  
 EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSLLW/PSLLD/PSLLQ—Bit Shift Left

| Opcode/<br>Instruction                                      | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F F1 /r<br>PSLLW xmm1, xmm2/m128                        | RM         | V/V                          | SSE2                     | Shift words in xmm1 left by amount specified in xmm2/m128 while shifting in 0s.       |
| 66 0F 71 /6 ib<br>PSLLW xmm1, imm8                          | MI         | V/V                          | SSE2                     | Shift words in xmm1 left by imm8 while shifting in 0s.                                |
| 66 0F 72 /6 ib<br>PSLLD xmm1, imm8                          | MI         | V/V                          | SSE2                     | Shift doublewords in xmm1 left by imm8 while shifting in 0s.                          |
| 66 0F 73 /6 ib<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>VPSLLQ ymm1, ymm2, imm8   | VMI        | V/V                          | AVX2                     | Shift quadwords in ymm2 left by imm8 while shifting in 0s.                            |



| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F.WIG F1 /r<br>VPSLLW xmm1 {k1}{z}, xmm2,<br>xmm3/m128           | M128       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 left by amount specified in<br>xmm3/m128 while shifting in 0s using writemask k1.          |
| EVEX.NDS.256.66.0F.WIG F1 /r<br>VPSLLW ymm1 {k1}{z}, ymm2,<br>xmm3/m128           | M128       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 left by amount specified in<br>xmm3/m128 while shifting in 0s using writemask k1.          |
| EVEX.NDS.512.66.0F.WIG F1 /r<br>VPSLLW zmm1 {k1}{z}, zmm2,<br>xmm3/m128           | M128       | V/V                          | AVX512BW                 | Shift words in zmm2 left by amount specified in<br>xmm3/m128 while shifting in 0s using writemask k1.          |
| EVEX.NDD.128.66.0F.WIG 71 /6 ib<br>VPSLLW xmm1 {k1}{z}, xmm2/m128,<br>imm8        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2/m128 left by imm8 while<br>shifting in 0s using writemask k1.                              |
| EVEX.NDD.256.66.0F.WIG 71 /6 ib<br>VPSLLW ymm1 {k1}{z}, ymm2/m256,<br>imm8        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2/m256 left by imm8 while<br>shifting in 0s using writemask k1.                              |
| EVEX.NDD.512.66.0F.WIG 71 /6 ib<br>VPSLLW zmm1 {k1}{z}, zmm2/m512,<br>imm8        | FVM        | V/V                          | AVX512BW                 | Shift words in zmm2/m512 left by imm8 while<br>shifting in 0 using writemask k1.                               |
| EVEX.NDS.128.66.0F.W0 F2 /r<br>VPSLLD xmm1 {k1}{z}, xmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 left by amount specified<br>in xmm3/m128 while shifting in 0s under writemask<br>k1. |
| EVEX.NDS.256.66.0F.W0 F2 /r<br>VPSLLD ymm1 {k1}{z}, ymm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 left by amount specified<br>in xmm3/m128 while shifting in 0s under writemask<br>k1. |
| EVEX.NDS.512.66.0F.W0 F2 /r<br>VPSLLD zmm1 {k1}{z}, zmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512F                  | Shift doublewords in zmm2 left by amount specified<br>in xmm3/m128 while shifting in 0s under writemask<br>k1. |
| EVEX.NDD.128.66.0F.W0 72 /6 ib<br>VPSLLD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2/m128/m32bcst left by<br>imm8 while shifting in 0s using writemask k1.                |
| EVEX.NDD.256.66.0F.W0 72 /6 ib<br>VPSLLD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2/m256/m32bcst left by<br>imm8 while shifting in 0s using writemask k1.                |
| EVEX.NDD.512.66.0F.W0 72 /6 ib<br>VPSLLD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | FVI        | V/V                          | AVX512F                  | Shift doublewords in zmm2/m512/m32bcst left by<br>imm8 while shifting in 0s using writemask k1.                |
| EVEX.NDS.128.66.0F.W1 F3 /r<br>VPSLLQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 left by amount specified in<br>xmm3/m128 while shifting in 0s using writemask k1.      |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDS.256.66.0F.W1 F3 /r<br>VPSLLQ ymm1 {k1}{z}, ymm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 left by amount specified in<br>xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDS.512.66.0F.W1 F3 /r<br>VPSLLQ zmm1 {k1}{z}, zmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512F                  | Shift quadwords in zmm2 left by amount specified in<br>xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDD.128.66.0F.W1 73 /6 ib<br>VPSLLQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2/m128/m64bcst left by<br>imm8 while shifting in 0s using writemask k1.             |
| EVEX.NDD.256.66.0F.W1 73 /6 ib<br>VPSLLQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2/m256/m64bcst left by<br>imm8 while shifting in 0s using writemask k1.             |
| EVEX.NDD.512.66.0F.W1 73 /6 ib<br>VPSLLQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | FVI        | V/V                          | AVX512F                  | Shift quadwords in zmm2/m512/m64bcst left by<br>imm8 while shifting in 0s using writemask k1.             |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| MI    | ModRM:r/m (r, w) | Imm8          | NA            | NA        |
| RM    | ModRM:reg (w)    | ModRM:r/m (r) | NA            | NA        |
| VMI   | VEX.vvvv (w)     | ModRM:r/m (R) | Imm8          | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8          | NA        |
| FVI   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8          | NA        |
| M128  | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |

### Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count 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.

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 PSLLW instruction shifts each of the words in the first source operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the first source operand; and the PSLLQ instruction shifts the quadword (or quadwords) in the first source 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: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

### Operation

```
LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] << COUNT);
FI;
```

```
LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;
```

```
LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] << COUNT);
FI;
```

```
LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
```

```

FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ←ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
    DEST[191:128] ←ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] << COUNT);
FI;
    
```

**VPSLLW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLW (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSLLW (ymm, imm8) - VEX**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORD\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSLLW (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSLLW (xmm, imm8) - VEX**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSLLW (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSLLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSLLD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+31:i] ← LOGICAL\_LEFT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[i+31:i] ← LOGICAL\_LEFT\_SHIFT\_DWORDS1(SRC1[i+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSLLD (ymm, ymm, xmm/m128) - VEX**

```

DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLD (ymm, imm8) - VEX**

```

DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLD (xmm, xmm, xmm/m128) - VEX**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPSLLD (xmm, imm8) - VEX**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ← 0

```

**PSLLD (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**PSLLD (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)

```

**VPSLLQ (EVEX versions, imm8)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
    ELSE DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR

```

```

    FI;
  ENDFOR

```

**VPSLLQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

```

    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)

```

FI;

IF VL = 256

```

    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)

```

FI;

IF VL = 512

```

    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)

```

```

    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)

```

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

```

    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]

```

ELSE

```

    IF *merging-masking* ; merging-masking

```

```

    THEN *DEST[i+63:i] remains unchanged*

```

```

    ELSE *zeroing-masking* ; zeroing-masking

```

```

    DEST[i+63:i] ← 0

```

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLQ (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

VPSLLQ (ymm, imm8) - VEX

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSLLQ (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSLLQ (xmm, imm8) - VEX**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSLLQ (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSLLQ (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)



## Intel C/C++ Compiler Intrinsic Equivalent

```

VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_slli_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW __m128i _mm_slli_epi16(__m128i m, int count)
PSLLW __m128i _mm_sll_epi16(__m128i m, __m128i count)
PSLLD __m128i _mm_slli_epi32(__m128i m, int count)
PSLLD __m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLQ __m128i _mm_slli_epi64(__m128i m, int count)
PSLLQ __m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLW __m256i _mm256_slli_epi16(__m256i m, int count)
VPSLLW __m256i _mm256_sll_epi16(__m256i m, __m128i count)
VPSLLD __m256i _mm256_slli_epi32(__m256i m, int count)
VPSLLD __m256i _mm256_sll_epi32(__m256i m, __m128i count)
VPSLLQ __m256i _mm256_slli_epi64(__m256i m, int count)

```

VPSLLQ \_\_m256i \_mm256\_sll\_epi64 (\_\_m256i m, \_\_m128i count)

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

VEX-encoded instructions:

- Syntax with RM/RVM operand encoding, see Exceptions Type 4.

- Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSLLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

- Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

- Syntax with FVI operand encoding, see Exceptions Type E4.

## PSRAW/PSRAD/PSRAQ—Bit Shift Arithmetic Right

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F E1 /r<br>PSRAW xmm1, xmm2/m128                                    | RM         | V/V                          | SSE2                     | Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits.                    |
| 66 0F 71 /4 ib<br>PSRAW xmm1, imm8                                      | MI         | V/V                          | SSE2                     | Shift words in xmm1 right by imm8 while shifting in sign bits.   |
| 66 0F E2 /r<br>PSRAD xmm1, xmm2/m128                                    | RM         | V/V                          | SSE2                     | Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits.              |
| 66 0F 72 /4 ib<br>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<br>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<br>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<br>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<br>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<br>VPSRAW ymm1, ymm2, ymm3/m128             | RVM        | V/V                          | AVX2                     | Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.                    |
| VEX.NDD.256.66.0F.WIG 71 /4 ib<br>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<br>VPSRAD ymm1, ymm2, ymm3/m128             | RVM        | V/V                          | AVX2                     | Shift doublewords in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.              |
| VEX.NDD.256.66.0F.WIG 72 /4 ib<br>VPSRAD ymm1, ymm2, imm8               | VMI        | V/V                          | AVX2                     | Shift doublewords in ymm2 right by imm8 while shifting in sign bits.                                       |
| EVEX.NDS.128.66.0F.WIG E1 /r<br>VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128    | M128       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1. |
| EVEX.NDS.256.66.0F.WIG E1 /r<br>VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128    | M128       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1. |
| EVEX.NDS.512.66.0F.WIG E1 /r<br>VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128    | M128       | V/V                          | AVX512BW                 | Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1. |
| EVEX.NDD.128.66.0F.WIG 71 /4 ib<br>VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.                     |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.NDD.256.66.0F.WIG 71 /4 ib<br>VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8           | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.                           |
| EVEX.NDD.512.66.0F.WIG 71 /4 ib<br>VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8           | FVM        | V/V                          | AVX512BW                 | Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.                           |
| EVEX.NDS.128.66.0F.W0 E2 /r<br>VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128               | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1. |
| EVEX.NDS.256.66.0F.W0 E2 /r<br>VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128               | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1. |
| EVEX.NDS.512.66.0F.W0 E2 /r<br>VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128               | M128       | V/V                          | AVX512F                  | Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1. |
| EVEX.NDD.128.66.0F.W0 72 /4 ib<br>VPSRAD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.             |
| EVEX.NDD.256.66.0F.W0 72 /4 ib<br>VPSRAD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.             |
| EVEX.NDD.512.66.0F.W0 72 /4 ib<br>VPSRAD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | FVI        | V/V                          | AVX512F                  | Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.             |
| EVEX.NDS.128.66.0F.W1 E2 /r<br>VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128               | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.   |
| EVEX.NDS.256.66.0F.W1 E2 /r<br>VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128               | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.   |
| EVEX.NDS.512.66.0F.W1 E2 /r<br>VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128               | M128       | V/V                          | AVX512F                  | Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.   |
| EVEX.NDD.128.66.0F.W1 72 /4 ib<br>VPSRAQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.               |
| EVEX.NDD.256.66.0F.W1 72 /4 ib<br>VPSRAQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | FVI        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.               |
| EVEX.NDD.512.66.0F.W1 72 /4 ib<br>VPSRAQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | FVI        | V/V                          | AVX512F                  | Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.               |

## Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| MI    | ModRM:r/m (r, w) | Imm8          | NA            | NA        |
| RM    | ModRM:reg (w)    | ModRM:r/m (r) | NA            | NA        |
| VMI   | VEX.vvvv (w)     | ModRM:r/m (R) | Imm8          | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8          | NA        |
| FVI   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8          | NA        |
| M128  | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count 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. 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 filled with the initial value of the sign bit.

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 first source operand to the right by the number of bits specified in the count 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: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

## Operation

```
ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC[63:0];
```

```
IF (COUNT > 31)
```

```
THEN
```

```
    DEST[31:0] ← SignBit
```

```
ELSE
```

```
    DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
```

```
FI;
```

```
ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC[63:0];
```

```
IF (COUNT > 63)
```

```

THEN
    DEST[63:0] ← SignBit
ELSE
    DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
FI;

```

```

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL)           ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT ← 64;
FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);

```

**VPSRAW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAW (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPSRAW (ymm, imm8) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0

**VPSRAW (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRAW (xmm, imm8) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRAW (xmm, xmm, xmm/m128)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRAW (xmm, imm8)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSRAD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+31:i] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[i+31:i] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC1[i+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0



**VPSRAD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAD (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPSRAD (ymm, imm8) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0

**VPSRAD (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRAD (xmm, imm8) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRAD (xmm, xmm, xmm/m128)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRAD (xmm, imm8)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSRAQ (EVEX versions, imm8)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[j+63:i] ← ARITHMETIC\_RIGHT\_SHIFT\_QWORDS1(SRC1[63:0], imm8)

ELSE DEST[j+63:i] ← ARITHMETIC\_RIGHT\_SHIFT\_QWORDS1(SRC1[j+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_DEST[VL-1:0] ← ARITHMETIC\_RIGHT\_SHIFT\_QWORDS(SRC1[VL-1:0], SRC2, VL)

FOR j ← 0 TO 7

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[j+63:i] ← TMP\_DEST[j+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRAD \_\_m512i \_mm512\_srai\_epi32(\_\_m512i a, unsigned int imm);

VPSRAD \_\_m512i \_mm512\_mask\_srai\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, unsigned int imm);

VPSRAD \_\_m512i \_mm512\_maskz\_srai\_epi32(\_\_mmask16 k, \_\_m512i a, unsigned int imm);

VPSRAD \_\_m256i \_mm256\_mask\_srai\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRAD \_\_m256i \_mm256\_maskz\_srai\_epi32(\_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRAD \_\_m128i \_mm\_mask\_srai\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRAD \_\_m128i \_mm\_maskz\_srai\_epi32(\_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRAD \_\_m512i \_mm512\_sra\_epi32(\_\_m512i a, \_\_m128i cnt);

VPSRAD \_\_m512i \_mm512\_mask\_sra\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);

VPSRAD \_\_m512i \_mm512\_maskz\_sra\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);

VPSRAD \_\_m256i \_mm256\_mask\_sra\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

VPSRAD \_\_m256i \_mm256\_maskz\_sra\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

VPSRAD \_\_m128i \_mm\_mask\_sra\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPSRAD \_\_m128i \_mm\_maskz\_sra\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPSRAQ \_\_m512i \_mm512\_srai\_epi64(\_\_m512i a, unsigned int imm);

VPSRAQ \_\_m512i \_mm512\_mask\_srai\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, unsigned int imm)

VPSRAQ \_\_m512i \_\_mm512\_maskz\_srai\_epi64( \_\_mmask8 k, \_\_m512i a, unsigned int imm)  
 VPSRAQ \_\_m256i \_\_mm256\_mask\_srai\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSRAQ \_\_m256i \_\_mm256\_maskz\_srai\_epi64( \_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSRAQ \_\_m128i \_\_mm\_mask\_srai\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAQ \_\_m128i \_\_mm\_maskz\_srai\_epi64( \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAQ \_\_m512i \_\_mm512\_sra\_epi64( \_\_m512i a, \_\_m128i cnt);  
 VPSRAQ \_\_m512i \_\_mm512\_mask\_sra\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m128i cnt)  
 VPSRAQ \_\_m512i \_\_mm512\_maskz\_sra\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m128i cnt)  
 VPSRAQ \_\_m256i \_\_mm256\_mask\_sra\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAQ \_\_m256i \_\_mm256\_maskz\_sra\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAQ \_\_m128i \_\_mm\_mask\_sra\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRAQ \_\_m128i \_\_mm\_maskz\_sra\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_srai\_epi16( \_\_m512i a, unsigned int imm);  
 VPSRAW \_\_m512i \_\_mm512\_mask\_srai\_epi16( \_\_m512i s, \_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSRAW \_\_m512i \_\_mm512\_maskz\_srai\_epi16( \_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSRAW \_\_m256i \_\_mm256\_mask\_srai\_epi16( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRAW \_\_m256i \_\_mm256\_maskz\_srai\_epi16( \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRAW \_\_m128i \_\_mm\_mask\_srai\_epi16( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAW \_\_m128i \_\_mm\_maskz\_srai\_epi16( \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAW \_\_m512i \_\_mm512\_sra\_epi16( \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_mask\_sra\_epi16( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_maskz\_sra\_epi16( \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m256i \_\_mm256\_mask\_sra\_epi16( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAW \_\_m256i \_\_mm256\_maskz\_sra\_epi16( \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAW \_\_m128i \_\_mm\_mask\_sra\_epi16( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRAW \_\_m128i \_\_mm\_maskz\_sra\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSRAW \_\_m128i \_\_mm\_srai\_epi16( \_\_m128i m, int count)  
 PSRAW \_\_m128i \_\_mm\_sra\_epi16( \_\_m128i m, \_\_m128i count)  
 VPSRAW \_\_m256i \_\_mm256\_sra\_epi16( \_\_m256i m, \_\_m128i count)  
 PSRAD \_\_m128i \_\_mm\_srai\_epi32( \_\_m128i m, int count)  
 PSRAD \_\_m128i \_\_mm\_sra\_epi32( \_\_m128i m, \_\_m128i count)  
 VPSRAD \_\_m256i \_\_mm256\_sra\_epi32( \_\_m256i m, \_\_m128i count)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRAW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

## PSRLDQ—Byte Shift Right

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F 73 /3 ib<br>PSRLDQ xmm1, imm8                              | MI         | V/V                          | SSE2                     | Shift xmm1 right by imm8 bytes while shifting in 0s and store result in xmm1.      |
| VEX.NDD.128.66.0F 73 /3 ib<br>VPSRLDQ xmm1, xmm2, imm8           | VMI        | V/V                          | AVX                      | Shift xmm2 right by imm8 bytes while shifting in 0s and store result in xmm1.      |
| VEX.NDD.256.66.0F 73 /3 ib<br>VPSRLDQ ymm1, ymm2, imm8           | VMI        | V/V                          | AVX2                     | Shift ymm2 right by imm8 bytes while shifting in 0s and store result in ymm1.      |
| EVEX.NDD.128.66.0F.WIG 73 /3 ib<br>VPSRLDQ xmm1, xmm2/m128, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1. |
| EVEX.NDD.256.66.0F.WIG 73 /3 ib<br>VPSRLDQ ymm1, ymm2/m256, imm8 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1. |
| EVEX.NDD.512.66.0F.WIG 73 /3 ib<br>VPSRLDQ zmm1, zmm2/m512, imm8 | FVM        | V/V                          | AVX512BW                 | Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1. |

### 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        |
| FVM   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8      | NA        |

### Description

Shifts the source operand to the right by the number of bytes specified in the count 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 source and destination operands are XMM registers. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register.

### Operation

#### VPSRLDQ (EVEX.U1.512 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST[127:0] ← SRC[127:0] >> (TEMP \* 8)

DEST[255:128] ← SRC[255:128] >> (TEMP \* 8)

DEST[383:256] ← SRC[383:256] >> (TEMP \* 8)

DEST[511:384] ← SRC[511:384] >> (TEMP \* 8)

DEST[MAX\_VL-1:512] ← 0;

**VPSRLDQ (VEX.256 and EVEX.256 encoded version)**

TEMP  $\leftarrow$  COUNT  
 IF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FI  
 DEST[127:0]  $\leftarrow$  SRC[127:0]  $\gg$  (TEMP \* 8)  
 DEST[255:128]  $\leftarrow$  SRC[255:128]  $\gg$  (TEMP \* 8)  
 DEST[MAX\_VL-1:256]  $\leftarrow$  0;

**VPSRLDQ (VEX.128 and EVEX.128 encoded version)**

TEMP  $\leftarrow$  COUNT  
 IF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FI  
 DEST  $\leftarrow$  SRC  $\gg$  (TEMP \* 8)  
 DEST[MAX\_VL-1:128]  $\leftarrow$  0;

**PSRLDQ(128-bit Legacy SSE version)**

TEMP  $\leftarrow$  COUNT  
 IF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FI  
 DEST  $\leftarrow$  DEST  $\gg$  (TEMP \* 8)  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PSRLDQ \_\_m128i \_mm\_srli\_si128 (\_\_m128i a, int imm)  
 VPSRLDQ \_\_m256i \_mm256\_bsrl\_i\_epi128 (\_\_m256i, const int)  
 VPSRLDQ \_\_m512i \_mm512\_bsrl\_i\_epi128 (\_\_m512i, int)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.  
 EVEX-encoded instruction, see Exceptions Type E4NF.nb.

**PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical**

| Opcode/<br>Instruction                                      | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F D1 /r<br>PSRLW xmm1, xmm2/m128                        | RM         | V/V                          | SSE2                     | Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.       |
| 66 0F 71 /2 ib<br>PSRLW xmm1, imm8                          | MI         | V/V                          | SSE2                     | Shift words in xmm1 right by imm8 while shifting in 0s.                                |
| 66 0F D2 /r<br>PSRLD xmm1, xmm2/m128                        | RM         | V/V                          | SSE2                     | Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 72 /2 ib<br>PSRLD xmm1, imm8                          | MI         | V/V                          | SSE2                     | Shift doublewords in xmm1 right by imm8 while shifting in 0s.                          |
| 66 0F D3 /r<br>PSRLQ xmm1, xmm2/m128                        | RM         | V/V                          | SSE2                     | Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.   |
| 66 0F 73 /2 ib<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>VPSRLD ymm1, ymm2, imm8   | VMI        | V/V                          | AVX2                     | Shift doublewords in ymm2 right by imm8 while shifting in 0s.                          |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.NDS.256.66.0F.WIG D3 /r<br>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<br>VPSRLQ ymm1, ymm2, imm8                         | VMI        | V/V                          | AVX2                     | Shift quadwords in ymm2 right by imm8 while shifting in 0s.   |
| EVEX.NDS.128.66.0F.WIG D1 /r<br>VPSRLW xmm1 {k1}{z}, xmm2,<br>xmm3/m128           | M128       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.       |
| EVEX.NDS.256.66.0F.WIG D1 /r<br>VPSRLW ymm1 {k1}{z}, ymm2,<br>xmm3/m128           | M128       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.       |
| EVEX.NDS.512.66.0F.WIG D1 /r<br>VPSRLW zmm1 {k1}{z}, zmm2,<br>xmm3/m128           | M128       | V/V                          | AVX512BW                 | Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.       |
| EVEX.NDD.128.66.0F.WIG 71 /2 ib<br>VPSRLW xmm1 {k1}{z}, xmm2/m128,<br>imm8        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.                           |
| EVEX.NDD.256.66.0F.WIG 71 /2 ib<br>VPSRLW ymm1 {k1}{z}, ymm2/m256,<br>imm8        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.                           |
| EVEX.NDD.512.66.0F.WIG 71 /2 ib<br>VPSRLW zmm1 {k1}{z}, zmm2/m512,<br>imm8        | FVM        | V/V                          | AVX512BW                 | Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.                           |
| EVEX.NDS.128.66.0F.W0 D2 /r<br>VPSRLD xmm1 {k1}{z}, xmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDS.256.66.0F.W0 D2 /r<br>VPSRLD ymm1 {k1}{z}, ymm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDS.512.66.0F.W0 D2 /r<br>VPSRLD zmm1 {k1}{z}, zmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512F                  | Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDD.128.66.0F.W0 72 /2 ib<br>VPSRLD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.             |
| EVEX.NDD.256.66.0F.W0 72 /2 ib<br>VPSRLD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.             |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDD.512.66.0F.W0 72 /2 ib<br>VPSRLD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | FVI        | V/V                          | AVX512F                  | Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.           |
| EVEX.NDS.128.66.0F.W1 D3 /r<br>VPSRLQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDS.256.66.0F.W1 D3 /r<br>VPSRLQ ymm1 {k1}{z}, ymm2,<br>xmm3/m128            | M128       | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDS.512.66.0F.W1 D3 /r<br>VPSRLQ zmm1 {k1}{z}, zmm2,<br>xmm3/m128            | M128       | V/V                          | AVX512F                  | Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1. |
| EVEX.NDD.128.66.0F.W1 73 /2 ib<br>VPSRLQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.             |
| EVEX.NDD.256.66.0F.W1 73 /2 ib<br>VPSRLQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.             |
| EVEX.NDD.512.66.0F.W1 73 /2 ib<br>VPSRLQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | FVI        | V/V                          | AVX512F                  | Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.             |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| MI    | ModRM:r/m (r, w) | Imm8          | NA            | NA        |
| RM    | ModRM:reg (w)    | ModRM:r/m (r) | NA            | NA        |
| VMI   | VEX.vvvv (w)     | ModRM:r/m (R) | Imm8          | NA        |
| RVM   | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |
| FVM   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8          | NA        |
| FVI   | EVEX.vvvv (w)    | ModRM:r/m (R) | Imm8          | NA        |
| M128  | ModRM:reg (w)    | VEX.vvvv (r)  | ModRM:r/m (r) | NA        |

### Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count 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.

The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the second source operand is a memory address, 128 bits are loaded. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.



The PSRLW instruction shifts each of the words in the first source operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the doublewords in the first source operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the first source 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 (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

### Operation

```
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC[63:0];
```

```
IF (COUNT > 31)
```

```
THEN
```

```
    DEST[31:0] ← 0
```

```
ELSE
```

```
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
```

```
FI;
```

```
LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC[63:0];
```

```
IF (COUNT > 63)
```

```
THEN
```

```
    DEST[63:0] ← 0
```

```
ELSE
```

```
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
```

```
FI;
```

```
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC[63:0];
```

```
IF (COUNT > 15)
```

```
THEN
```

```
    DEST[255:0] ← 0
```

```
ELSE
```

```
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
```

```
    (* Repeat shift operation for 2nd through 15th words *)
```

```
    DEST[255:240] ← ZeroExtend(SRC[255:240] >> COUNT);
```

```
FI;
```

```
LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
```

```
COUNT ← COUNT_SRC[63:0];
```

```

IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ← ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
FI;

```

**VPSRLW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLW (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLW (ymm, imm8) - VEX**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLW (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRLW (xmm, imm8) - VEX**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRLW (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSRLD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSRLD (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLD (ymm, imm8) - VEX**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLD (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRLD (xmm, imm8) - VEX**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRLD (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRLD (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSRLQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLQ (EVEX versions, imm8)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS1(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS1(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLQ (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLQ (ymm, imm8) - VEX**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLQ (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRLQ (xmm, imm8) - VEX**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRLQ (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRLQ (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRLD \_\_m512i \_mm512\_srl\_epi32(\_\_m512i a, unsigned int imm);  
 VPSRLD \_\_m512i \_mm512\_mask\_srl\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, unsigned int imm);  
 VPSRLD \_\_m512i \_mm512\_maskz\_srl\_epi32(\_\_mmask16 k, \_\_m512i a, unsigned int imm);  
 VPSRLD \_\_m256i \_mm256\_mask\_srl\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSRLD \_\_m256i \_mm256\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSRLD \_\_m128i \_mm\_mask\_srl\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRLD \_\_m128i \_mm\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRLD \_\_m512i \_mm512\_srl\_epi32(\_\_m512i a, \_\_m128i cnt);  
 VPSRLD \_\_m512i \_mm512\_mask\_srl\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRLD \_\_m512i \_mm512\_maskz\_srl\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRLD \_\_m256i \_mm256\_mask\_srl\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRLD \_\_m256i \_mm256\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRLD \_\_m128i \_mm\_mask\_srl\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLD \_\_m128i \_mm\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLQ \_\_m512i \_mm512\_srl\_epi64(\_\_m512i a, unsigned int imm);  
 VPSRLQ \_\_m512i \_mm512\_mask\_srl\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, unsigned int imm);  
 VPSRLQ \_\_m512i \_mm512\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m512i a, unsigned int imm);  
 VPSRLQ \_\_m256i \_mm256\_mask\_srl\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSRLQ \_\_m256i \_mm256\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSRLQ \_\_m128i \_mm\_mask\_srl\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRLQ \_\_m128i \_mm\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRLQ \_\_m512i \_mm512\_srl\_epi64(\_\_m512i a, \_\_m128i cnt);  
 VPSRLQ \_\_m512i \_mm512\_mask\_srl\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRLQ \_\_m512i \_mm512\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRLQ \_\_m256i \_mm256\_mask\_srl\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRLQ \_\_m256i \_mm256\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRLQ \_\_m128i \_mm\_mask\_srl\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLQ \_\_m128i \_mm\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLW \_\_m512i \_mm512\_srl\_epi16(\_\_m512i a, unsigned int imm);  
 VPSRLW \_\_m512i \_mm512\_mask\_srl\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSRLW \_\_m512i \_mm512\_maskz\_srl\_epi16(\_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSRLW \_\_m256i \_mm256\_mask\_srl\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRLW \_\_m256i \_mm256\_maskz\_srl\_epi16(\_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRLW \_\_m128i \_mm\_mask\_srl\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRLW \_\_m128i \_mm\_maskz\_srl\_epi16(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRLW \_\_m512i \_mm512\_srl\_epi16(\_\_m512i a, \_\_m128i cnt);  
 VPSRLW \_\_m512i \_mm512\_mask\_srl\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);

VPSRLW \_\_m512i \_\_mm512\_maskz\_srl\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRLW \_\_m256i \_\_mm256\_mask\_srl\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRLW \_\_m256i \_\_mm256\_maskz\_srl\_epi16(\_\_mmask8 k, \_\_mmask16 a, \_\_m128i cnt);  
 VPSRLW \_\_m128i \_\_mm\_mask\_srl\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLW \_\_m128i \_\_mm\_maskz\_srl\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSRLW \_\_m128i \_\_mm\_srl\_epi16(\_\_m128i m, int count)  
 PSRLW \_\_m128i \_\_mm\_srl\_epi16(\_\_m128i m, \_\_m128i count)  
 VPSRLW \_\_m256i \_\_mm256\_srl\_epi16(\_\_m256i m, \_\_m128i count)  
 PSRLD \_\_m128i \_\_mm\_srl\_epi32(\_\_m128i m, int count)  
 PSRLD \_\_m128i \_\_mm\_srl\_epi32(\_\_m128i m, \_\_m128i count)  
 VPSRLD \_\_m256i \_\_mm256\_srl\_epi32(\_\_m256i m, \_\_m128i count)  
 PSRLQ \_\_m128i \_\_mm\_srl\_epi64(\_\_m128i m, int count)  
 PSRLQ \_\_m128i \_\_mm\_srl\_epi64(\_\_m128i m, \_\_m128i count)  
 VPSRLQ \_\_m256i \_\_mm256\_srl\_epi64(\_\_m256i m, \_\_m128i count)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.



## VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W0 47 /r<br>VPSLLVD xmm1, xmm2, xmm3/m128                     | RVM        | V/V                          | AVX2                     | Shift 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<br>VPSLLVQ xmm1, xmm2, xmm3/m128                     | RVM        | V/V                          | AVX2                     | Shift 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<br>VPSLLVD ymm1, ymm2, ymm3/m256                     | RVM        | V/V                          | AVX2                     | Shift 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<br>VPSLLVQ ymm1, ymm2, ymm3/m256                     | RVM        | V/V                          | AVX2                     | Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.                              |
| EVEX.NDS.128.66.0F38.W1 12 /r<br>VPSLLVW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.               |
| EVEX.NDS.256.66.0F38.W1 12 /r<br>VPSLLVW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.               |
| EVEX.NDS.512.66.0F38.W1 12 /r<br>VPSLLVW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | FVM        | V/V                          | AVX512BW                 | Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.               |
| EVEX.NDS.128.66.0F38.W0 47 /r<br>VPSLLVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1. |
| EVEX.NDS.256.66.0F38.W0 47 /r<br>VPSLLVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 47 /r<br>VPSLLVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1. |
| EVEX.NDS.128.66.0F38.W1 47 /r<br>VPSLLVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.NDS.256.66.0F38.W1 47 /r<br>VPSLLVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.NDS.512.66.0F38.W1 47 /r<br>VPSLLVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.   |

## 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        |
| FVM   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Shifts the bits in the individual data elements (words, 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 15 (for word), 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 (MAX\_VL-1:128) of the corresponding destination 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. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

## Operation

**VPSLLVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**VPSLLVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 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[MAX_VL-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[228 : 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;
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
      ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**VPSLLVQ (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 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ← 0;
DEST[MAX_VL-1:128] ← 0;

```

**VPSLLVQ (VEX.256 version)**

```

COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 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 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] ← 0;
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLVQ (EVEX encoded version)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
      ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSLLVW __m512i __mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i __mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i __mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i __mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i __mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i __mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);

```

VPSLLVW \_\_m128i \_mm\_maskz\_sllv\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m512i \_mm512\_sllv\_epi32(\_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m512i \_mm512\_mask\_sllv\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m512i \_mm512\_maskz\_sllv\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m256i \_mm256\_mask\_sllv\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m256i \_mm256\_maskz\_sllv\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m128i \_mm\_mask\_sllv\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m128i \_mm\_maskz\_sllv\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVQ \_\_m512i \_mm512\_sllv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPSLLVQ \_\_m512i \_mm512\_mask\_sllv\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVQ \_\_m512i \_mm512\_maskz\_sllv\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m256i \_mm256\_mask\_sllv\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m256i \_mm256\_maskz\_sllv\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m128i \_mm\_mask\_sllv\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m128i \_mm\_maskz\_sllv\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m256i \_mm256\_sllv\_epi32 (\_\_m256i m, \_\_m256i count)  
 VPSLLVQ \_\_m256i \_mm256\_sllv\_epi64 (\_\_m256i m, \_\_m256i count)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.

EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.

## VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38.W0 45 /r<br>VPSRLVD xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX2                     | Shift 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<br>VPSRLVQ xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX2                     | Shift 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<br>VPSRLVD ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Shift 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<br>VPSRLVQ ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.                              |
| EVEX.NDS.128.66.0F38.W1 10 /r<br>VPSRLVW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.               |
| EVEX.NDS.256.66.0F38.W1 10 /r<br>VPSRLVW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.               |
| EVEX.NDS.512.66.0F38.W1 10 /r<br>VPSRLVW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | FVM       | V/V                          | AVX512BW                 | Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.               |
| EVEX.NDS.128.66.0F38.W0 45 /r<br>VPSRLVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1. |
| EVEX.NDS.256.66.0F38.W0 45 /r<br>VPSRLVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 45 /r<br>VPSRLVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1. |
| EVEX.NDS.128.66.0F38.W1 45 /r<br>VPSRLVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.NDS.256.66.0F38.W1 45 /r<br>VPSRLVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.NDS.512.66.0F38.W1 45 /r<br>VPSRLVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.   |

## 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        |
| FVM   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Shifts the bits in the individual data elements (words, 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 15 (for word), 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 (MAX\_VL-1:128) of the corresponding destination 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. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

## Operation

**VPSRLVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] >> SRC2[j+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**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[MAX_VL-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;
DEST[MAX_VL-1:256] ← 0;

```

**VPSRLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
            ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 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 < 64 THEN
    DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] ← 0;
DEST[MAX_VL-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 < 64 THEN
    DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] ← 0;
DEST[MAX_VL-1:256] ← 0;

```

**VPSRLVQ (EVEX encoded version)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
            ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSRLVW __m512i __mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i __mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i __mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i __mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i __mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i __mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);

```

VPSRLVW \_\_m128i \_mm\_maskz\_srlv\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVW \_\_m256i \_mm256\_srlv\_epi32( \_\_m256i m, \_\_m256i count)  
 VPSRLVD \_\_m512i \_mm512\_srlv\_epi32( \_\_m512i a, \_\_m512i cnt);  
 VPSRLVD \_\_m512i \_mm512\_mask\_srlv\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVD \_\_m512i \_mm512\_maskz\_srlv\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVD \_\_m256i \_mm256\_mask\_srlv\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVD \_\_m256i \_mm256\_maskz\_srlv\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVD \_\_m128i \_mm\_mask\_srlv\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVD \_\_m128i \_mm\_maskz\_srlv\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_srlv\_epi64( \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_mask\_srlv\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_maskz\_srlv\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_mask\_srlv\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_maskz\_srlv\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVQ \_\_m128i \_mm\_mask\_srlv\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m128i \_mm\_maskz\_srlv\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_srlv\_epi64( \_\_m256i m, \_\_m256i count)  
 VPSRLVD \_\_m128i \_mm\_srlv\_epi32( \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m128i \_mm\_srlv\_epi64( \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.

EVEX-encoded VPSRLVW, see Exceptions Type E4.nb.

## PSUBB/PSUBW/PSUBD/PSUBQ—Packed Integer Subtract

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| 66 0F F8 /r<br>PSUBB xmm1, xmm2/m128                                    | RM         | V/V                          | SSE2                     | Subtract packed byte integers in xmm2/m128 from xmm1.                                      |
| 66 0F F9 /r<br>PSUBW xmm1, xmm2/m128                                    | RM         | V/V                          | SSE2                     | Subtract packed word integers in xmm2/m128 from xmm1.                                      |
| 66 0F FA /r<br>PSUBD xmm1, xmm2/m128                                    | RM         | V/V                          | SSE2                     | Subtract packed doubleword integers in xmm2/m128 from xmm1.                                |
| 66 0F FB/r<br>PSUBQ xmm1, xmm2/m128                                     | RM         | V/V                          | SSE2                     | Subtract packed quadword integers in xmm2/m128 from xmm1.                                  |
| VEX.NDS.128.66.0F.WIG F8 /r<br>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<br>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<br>VPSUBD xmm1, xmm2, xmm3/m128             | RVM        | V/V                          | AVX                      | Subtract packed doubleword integers in xmm3/m128 from xmm2.                                |
| VEX.NDS.128.66.0F.WIG FB/r<br>VPSUBQ xmm1, xmm2, xmm3/m128              | RVM        | V/V                          | AVX                      | Subtract packed quadword integers in xmm3/m128 from xmm2.                                  |
| VEX.NDS.256.66.0F.WIG F8 /r<br>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<br>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<br>VPSUBD ymm1, ymm2, ymm3/m256             | RVM        | V/V                          | AVX2                     | Subtract packed doubleword integers in ymm3/m256 from ymm2.                                |
| VEX.NDS.256.66.0F.WIG FB/r<br>VPSUBQ ymm1, ymm2, ymm3/m256              | RVM        | V/V                          | AVX2                     | Subtract packed quadword integers in ymm3/m256 from ymm2.                                  |
| EVEX.NDS.128.66.0F.WIG F8 /r<br>VPSUBB xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F.WIG F8 /r<br>VPSUBB ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F.WIG F8 /r<br>VPSUBB zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM        | V/V                          | AVX512BW                 | Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F.WIG F9 /r<br>VPSUBW xmm1 {k1}{z}, xmm2,<br>xmm3/m128        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed word integers in xmm3/m128 from<br>xmm2 and store in xmm1 using writemask k1.                  |
| EVEX.NDS.256.66.0F.WIG F9 /r<br>VPSUBW ymm1 {k1}{z}, ymm2,<br>ymm3/m256        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed word integers in ymm3/m256 from<br>ymm2 and store in ymm1 using writemask k1.                  |
| EVEX.NDS.512.66.0F.WIG F9 /r<br>VPSUBW zmm1 {k1}{z}, zmm2,<br>zmm3/m512        | FVM        | V/V                          | AVX512BW                 | Subtract packed word integers in zmm3/m512 from<br>zmm2 and store in zmm1 using writemask k1.                  |
| EVEX.NDS.128.66.0F.W0 FA /r<br>VPSUBD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Subtract packed doubleword integers in<br>xmm3/m128/m32bcst from xmm2 and store in xmm1<br>using writemask k1. |
| EVEX.NDS.256.66.0F.W0 FA /r<br>VPSUBD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Subtract packed doubleword integers in<br>ymm3/m256/m32bcst from ymm2 and store in ymm1<br>using writemask k1. |
| EVEX.NDS.512.66.0F.W0 FA /r<br>VPSUBD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Subtract packed doubleword integers in<br>zmm3/m512/m32bcst from zmm2 and store in zmm1<br>using writemask k1  |
| EVEX.NDS.128.66.0F.W1 FB /r<br>VPSUBQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Subtract packed quadword integers in<br>xmm3/m128/m64bcst from xmm2 and store in xmm1<br>using writemask k1.   |
| EVEX.NDS.256.66.0F.W1 FB /r<br>VPSUBQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Subtract packed quadword integers in<br>ymm3/m256/m64bcst from ymm2 and store in ymm1<br>using writemask k1.   |
| EVEX.NDS.512.66.0F.W1 FB/r<br>VPSUBQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst  | FV         | V/V                          | AVX512F                  | Subtract packed quadword integers in<br>zmm3/m512/m64bcst from zmm2 and store in zmm1<br>using writemask k1.   |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Subtracts the packed byte, word, doubleword, or quadword integers in the second source operand from the first source operand and stores the result in the destination operand. When a result is too large to be represented in the 8/16/32/64 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 on which the values are operated.

128-bit Legacy SSE version: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD/Q: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### VPSUBB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC1[i+7:i] - SRC2[i+7:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+7:i] = 0
  FI
FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

### VPSUBW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC1[i+15:i] - SRC2[i+15:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+15:i] = 0
  FI
FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**VPSUBD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0]

ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPSUBQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPSUBB (VEX.256 encoded version)**

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]

DEST[15:8] ← SRC1[15:8]-SRC2[15:8]

DEST[23:16] ← SRC1[23:16]-SRC2[23:16]

DEST[31:24] ← SRC1[31:24]-SRC2[31:24]

DEST[39:32] ← SRC1[39:32]-SRC2[39:32]

DEST[47:40] ← SRC1[47:40]-SRC2[47:40]

DEST[55:48] ← SRC1[55:48]-SRC2[55:48]

DEST[63:56] ← SRC1[63:56]-SRC2[63:56]

DEST[71:64] ← SRC1[71:64]-SRC2[71:64]

DEST[79:72] ← SRC1[79:72]-SRC2[79:72]

DEST[87:80] ← SRC1[87:80]-SRC2[87:80]

DEST[95:88] ← SRC1[95:88]-SRC2[95:88]

DEST[103:96] ← SRC1[103:96]-SRC2[103:96]

DEST[111:104] ← SRC1[111:104]-SRC2[111:104]

DEST[119:112] ← SRC1[119:112]-SRC2[119:112]

DEST[127:120] ← SRC1[127:120]-SRC2[127:120]  
 DEST[135:128] ← SRC1[135:128]-SRC2[135:128]  
 DEST[143:136] ← SRC1[143:136]-SRC2[143:136]  
 DEST[151:144] ← SRC1[151:144]-SRC2[151:144]  
 DEST[159:152] ← SRC1[159:152]-SRC2[159:152]  
 DEST[167:160] ← SRC1[167:160]-SRC2[167:160]  
 DEST[175:168] ← SRC1[175:168]-SRC2[175:168]  
 DEST[183:176] ← SRC1[183:176]-SRC2[183:176]  
 DEST[191:184] ← SRC1[191:184]-SRC2[191:184]  
 DEST[199:192] ← SRC1[199:192]-SRC2[199:192]  
 DEST[207:200] ← SRC1[207:200]-SRC2[207:200]  
 DEST[215:208] ← SRC1[215:208]-SRC2[215:208]  
 DEST[223:216] ← SRC1[223:216]-SRC2[223:216]  
 DEST[231:224] ← SRC1[231:224]-SRC2[231:224]  
 DEST[239:232] ← SRC1[239:232]-SRC2[239:232]  
 DEST[247:240] ← SRC1[247:240]-SRC2[247:240]  
 DEST[255:248] ← SRC1[255:248]-SRC2[255:248]  
 DEST[MAX\_VL-1:256] ← 0

**VPSUBB (VEX.128 encoded version)**

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]  
 DEST[15:8] ← SRC1[15:8]-SRC2[15:8]  
 DEST[23:16] ← SRC1[23:16]-SRC2[23:16]  
 DEST[31:24] ← SRC1[31:24]-SRC2[31:24]  
 DEST[39:32] ← SRC1[39:32]-SRC2[39:32]  
 DEST[47:40] ← SRC1[47:40]-SRC2[47:40]  
 DEST[55:48] ← SRC1[55:48]-SRC2[55:48]  
 DEST[63:56] ← SRC1[63:56]-SRC2[63:56]  
 DEST[71:64] ← SRC1[71:64]-SRC2[71:64]  
 DEST[79:72] ← SRC1[79:72]-SRC2[79:72]  
 DEST[87:80] ← SRC1[87:80]-SRC2[87:80]  
 DEST[95:88] ← SRC1[95:88]-SRC2[95:88]  
 DEST[103:96] ← SRC1[103:96]-SRC2[103:96]  
 DEST[111:104] ← SRC1[111:104]-SRC2[111:104]  
 DEST[119:112] ← SRC1[119:112]-SRC2[119:112]  
 DEST[127:120] ← SRC1[127:120]-SRC2[127:120]  
 DEST[MAX\_VL-1:128] ← 0

**PSUBB (128-bit Legacy SSE version)**

DEST[7:0] ← DEST[7:0]-SRC[7:0]  
 DEST[15:8] ← DEST[15:8]-SRC[15:8]  
 DEST[23:16] ← DEST[23:16]-SRC[23:16]  
 DEST[31:24] ← DEST[31:24]-SRC[31:24]  
 DEST[39:32] ← DEST[39:32]-SRC[39:32]  
 DEST[47:40] ← DEST[47:40]-SRC[47:40]  
 DEST[55:48] ← DEST[55:48]-SRC[55:48]  
 DEST[63:56] ← DEST[63:56]-SRC[63:56]  
 DEST[71:64] ← DEST[71:64]-SRC[71:64]  
 DEST[79:72] ← DEST[79:72]-SRC[79:72]  
 DEST[87:80] ← DEST[87:80]-SRC[87:80]  
 DEST[95:88] ← DEST[95:88]-SRC[95:88]  
 DEST[103:96] ← DEST[103:96]-SRC[103:96]  
 DEST[111:104] ← DEST[111:104]-SRC[111:104]  
 DEST[119:112] ← DEST[119:112]-SRC[119:112]

DEST[127:120] ←DEST[127:120]-SRC[127:120]  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPSUBW (VEX.256 encoded version)**

DEST[15:0] ←SRC1[15:0]-SRC2[15:0]  
 DEST[31:16] ←SRC1[31:16]-SRC2[31:16]  
 DEST[47:32] ←SRC1[47:32]-SRC2[47:32]  
 DEST[63:48] ←SRC1[63:48]-SRC2[63:48]  
 DEST[79:64] ←SRC1[79:64]-SRC2[79:64]  
 DEST[95:80] ←SRC1[95:80]-SRC2[95:80]  
 DEST[111:96] ←SRC1[111:96]-SRC2[111:96]  
 DEST[127:112] ←SRC1[127:112]-SRC2[127:112]  
 DEST[143:128] ←SRC1[143:128]-SRC2[143:128]  
 DEST[159:144] ←SRC1[159:144]-SRC2[159:144]  
 DEST[175:160] ←SRC1[175:160]-SRC2[175:160]  
 DEST[191:176] ←SRC1[191:176]-SRC2[191:176]  
 DEST[207:192] ←SRC1[207:192]-SRC2[207:192]  
 DEST[223:208] ←SRC1[223:208]-SRC2[223:208]  
 DEST[239:224] ←SRC1[239:224]-SRC2[239:224]  
 DEST[255:240] ←SRC1[255:240]-SRC2[255:240]  
 DEST[MAX\_VL-1:256] ←0

**VPSUBW (VEX.128 encoded version)**

DEST[15:0] ←SRC1[15:0]-SRC2[15:0]  
 DEST[31:16] ←SRC1[31:16]-SRC2[31:16]  
 DEST[47:32] ←SRC1[47:32]-SRC2[47:32]  
 DEST[63:48] ←SRC1[63:48]-SRC2[63:48]  
 DEST[79:64] ←SRC1[79:64]-SRC2[79:64]  
 DEST[95:80] ←SRC1[95:80]-SRC2[95:80]  
 DEST[111:96] ←SRC1[111:96]-SRC2[111:96]  
 DEST[127:112] ←SRC1[127:112]-SRC2[127:112]  
 DEST[MAX\_VL-1:128] ←0

**PSUBW (128-bit Legacy SSE version)**

DEST[15:0] ←DEST[15:0]-SRC[15:0]  
 DEST[31:16] ←DEST[31:16]-SRC[31:16]  
 DEST[47:32] ←DEST[47:32]-SRC[47:32]  
 DEST[63:48] ←DEST[63:48]-SRC[63:48]  
 DEST[79:64] ←DEST[79:64]-SRC[79:64]  
 DEST[95:80] ←DEST[95:80]-SRC[95:80]  
 DEST[111:96] ←DEST[111:96]-SRC[111:96]  
 DEST[127:112] ←DEST[127:112]-SRC[127:112]  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPSUBD (VEX.256 encoded version)**

DEST[31:0] ←SRC1[31:0]-SRC2[31:0]  
 DEST[63:32] ←SRC1[63:32]-SRC2[63:32]  
 DEST[95:64] ←SRC1[95:64]-SRC2[95:64]  
 DEST[127:96] ←SRC1[127:96]-SRC2[127:96]  
 DEST[159:128] ←SRC1[159:128]-SRC2[159:128]  
 DEST[191:160] ←SRC1[191:160]-SRC2[191:160]  
 DEST[223:192] ←SRC1[223:192]-SRC2[223:192]  
 DEST[255:224] ←SRC1[255:224]-SRC2[255:224]  
 DEST[MAX\_VL-1:256] ←0



**VPSUBD (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32]-SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64]-SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96]-SRC2[127:96]  
 DEST[MAX\_VL-1:128] ← 0

**PSUBD (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0]-SRC[31:0]  
 DEST[63:32] ← DEST[63:32]-SRC[63:32]  
 DEST[95:64] ← DEST[95:64]-SRC[95:64]  
 DEST[127:96] ← DEST[127:96]-SRC[127:96]  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPSUBQ (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0]-SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64]-SRC2[127:64]  
 DEST[191:128] ← SRC1[191:128]-SRC2[191:128]  
 DEST[255:192] ← SRC1[255:192]-SRC2[255:192]  
 DEST[MAX\_VL-1:256] ← 0

**VPSUBQ (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0]-SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64]-SRC2[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**PSUBQ (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0]-SRC[63:0]  
 DEST[127:64] ← DEST[127:64]-SRC[127:64]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSUBB \_\_m512i\_mm512\_sub\_epi8(\_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m512i\_mm512\_mask\_sub\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m512i\_mm512\_maskz\_sub\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m256i\_mm256\_mask\_sub\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBB \_\_m256i\_mm256\_maskz\_sub\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBB \_\_m128i\_mm\_mask\_sub\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBB \_\_m128i\_mm\_maskz\_sub\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBW \_\_m512i\_mm512\_sub\_epi16(\_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m512i\_mm512\_mask\_sub\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m512i\_mm512\_maskz\_sub\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m256i\_mm256\_mask\_sub\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBW \_\_m256i\_mm256\_maskz\_sub\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBW \_\_m128i\_mm\_mask\_sub\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBW \_\_m128i\_mm\_maskz\_sub\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBD \_\_m512i\_mm512\_sub\_epi32(\_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m512i\_mm512\_mask\_sub\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m512i\_mm512\_maskz\_sub\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m256i\_mm256\_mask\_sub\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBD \_\_m256i\_mm256\_maskz\_sub\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBD \_\_m128i\_mm\_mask\_sub\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBD \_\_m128i\_mm\_maskz\_sub\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBQ \_\_m512i\_mm512\_sub\_epi64(\_\_m512i a, \_\_m512i b);

VPSUBQ \_\_m512i \_\_mm512\_mask\_sub\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPSUBQ \_\_m512i \_\_mm512\_maskz\_sub\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPSUBQ \_\_m256i \_\_mm256\_mask\_sub\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBQ \_\_m256i \_\_mm256\_maskz\_sub\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBQ \_\_m128i \_\_mm\_mask\_sub\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBQ \_\_m128i \_\_mm\_maskz\_sub\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBB \_\_m128i \_\_mm\_sub\_epi8 (\_\_m128i a, \_\_m128i b)  
 PSUBW \_\_m128i \_\_mm\_sub\_epi16 (\_\_m128i a, \_\_m128i b)  
 PSUBD \_\_m128i \_\_mm\_sub\_epi32 (\_\_m128i a, \_\_m128i b)  
 PSUBQ \_\_m128i \_\_mm\_sub\_epi64(\_\_m128i m1, \_\_m128i m2)  
 VPSUBB \_\_m256i \_\_mm256\_sub\_epi8 (\_\_m256i a, \_\_m256i b)  
 VPSUBW \_\_m256i \_\_mm256\_sub\_epi16 (\_\_m256i a, \_\_m256i b)  
 VPSUBD \_\_m256i \_\_mm256\_sub\_epi32 (\_\_m256i a, \_\_m256i b)  
 VPSUBQ \_\_m256i \_\_mm256\_sub\_epi64(\_\_m256i m1, \_\_m256i m2)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBD/Q, see Exceptions Type E4.

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F E8 /r<br>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.                                      |
| 66 0F E9 /r<br>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 E8 /r<br>VPSUBSB xmm1, xmm2,<br>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 E9 /r<br>VPSUBSW xmm1, xmm2,<br>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 E8 /r<br>VPSUBSB ymm1, ymm2,<br>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 E9 /r<br>VPSUBSW ymm1, ymm2,<br>ymm3/m256              | RVM       | V/V                          | AVX2                     | Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.                                      |
| EVEX.NDS.128.66.0F.WIG E8 /r<br>VPSUBSB xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F.WIG E8 /r<br>VPSUBSB ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F.WIG E8 /r<br>VPSUBSB zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1. |
| EVEX.NDS.128.66.0F.WIG E9 /r<br>VPSUBSW xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F.WIG E9 /r<br>VPSUBSW ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1. |
| EVEX.NDS.512.66.0F.WIG E9 /r<br>VPSUBSW zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1. |

## 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Subtract packed signed byte/word integers of the second source operand from the packed signed byte/word integers of the first source operand, and stores the packed integer results in the destination operand. An overflowed result larger than byte/word size 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.

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 (MAX\_VL-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 (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

**VPSUBSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8;

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+7:i] ← 0;

  FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPSUBSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0;

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;

**VPSUBSB (VEX.256 encoded version)**

DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);

(\* Repeat subtract operation for 2nd through 31th bytes \*)

DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);

DEST[MAX\_VL-1:256] ← 0;

**VPSUBSB (VEX.128 encoded version)**

DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);

(\* Repeat subtract operation for 2nd through 14th bytes \*)

DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);

DEST[MAX\_VL-1:128] ← 0;

**PSUBSB (128-bit Legacy SSE Version)**

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);

(\* Repeat subtract operation for 2nd through 14th bytes \*)

DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);

DEST[MAX\_VL-1:128] (Unmodified);

**VPSUBSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);

(\* Repeat subtract operation for 2nd through 15th words \*)

DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);

DEST[MAX\_VL-1:256] ← 0;

**VPSUBSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);

(\* Repeat subtract operation for 2nd through 7th words \*)

DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);

DEST[MAX\_VL-1:128] ← 0;

**PSUBSW (128-bit Legacy SSE Version)**

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);

(\* Repeat subtract operation for 2nd through 7th words \*)

DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);

DEST[MAX\_VL-1:128] (Unmodified);

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSUBSB \_\_m512i \_mm512\_subs\_epi8(\_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m512i \_mm512\_mask\_subs\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m512i \_mm512\_maskz\_subs\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m256i \_mm256\_mask\_subs\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSB \_\_m256i \_mm256\_maskz\_subs\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSB \_\_m128i \_mm\_mask\_subs\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSB \_\_m128i \_mm\_maskz\_subs\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSW \_\_m512i \_mm512\_subs\_epi16(\_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m512i \_mm512\_mask\_subs\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m512i \_mm512\_maskz\_subs\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m256i \_mm256\_mask\_subs\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSW \_\_m256i \_mm256\_maskz\_subs\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSW \_\_m128i \_mm\_mask\_subs\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSW \_\_m128i \_mm\_maskz\_subs\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBSB \_\_m128i \_mm\_subs\_epi8(\_\_m128i m1, \_\_m128i m2)  
 PSUBSW \_\_m128i \_mm\_subs\_epi16(\_\_m128i m1, \_\_m128i m2)  
 VPSUBSB \_\_m256i \_mm\_subs\_epi8(\_\_m256i m1, \_\_m256i m2)  
 VPSUBSW \_\_m256i \_mm\_subs\_epi16(\_\_m256i m1, \_\_m256i m2)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| 66 0F D8 /r<br>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.                                       |
| 66 0F D9 /r<br>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 D8 /r<br>VPSUBUSB xmm1, xmm2,<br>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 D9 /r<br>VPSUBUSW xmm1, xmm2,<br>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 D8 /r<br>VPSUBUSB ymm1, ymm2,<br>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 D9 /r<br>VPSUBUSW ymm1, ymm2,<br>ymm3/m256              | RVM       | V/V                          | AVX2                     | Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2 and saturate result.                                       |
| EVEX.NDS.128.66.0F.WIG D8 /r<br>VPSUBUSB xmm1 {k1}{z},<br>xmm2, xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.    |
| EVEX.NDS.256.66.0F.WIG D8 /r<br>VPSUBUSB ymm1 {k1}{z},<br>ymm2, ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.    |
| EVEX.NDS.512.66.0F.WIG D8 /r<br>VPSUBUSB zmm1 {k1}{z},<br>zmm2, zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.    |
| EVEX.NDS.128.66.0F.WIG D9 /r<br>VPSUBUSW xmm1 {k1}{z},<br>xmm2, xmm3/m128 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1. |
| EVEX.NDS.256.66.0F.WIG D9 /r<br>VPSUBUSW ymm1 {k1}{z},<br>ymm2, ymm3/m256 | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.    |
| EVEX.NDS.512.66.0F.WIG D9 /r<br>VPSUBUSW zmm1 {k1}{z},<br>zmm2, zmm3/m512 | FVM       | V/V                          | AVX512BW                 | Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.    |

## 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Subtract packed unsigned byte/word integers of the second source operand from the packed unsigned byte/word integers of the first source operand and stores the packed unsigned integer results in the destination operand. An overflowed result larger than byte/word size is handled with unsigned saturation, as described in the following paragraphs.

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.

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 (MAX\_VL-1:128) of the corresponding 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 (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

**VPSUBUSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8;

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+7:i] ← 0;

  FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;



**VPSUBUSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16;

IF k1[j] OR \*no writemask\*

THEN DEST[j+15:i] ← SaturateToUnsignedWord (SRC1[j+15:i] - SRC2[j+15:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+15:i] ← 0;

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;

**VPSUBUSB (VEX.256 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);

(\* Repeat subtract operation for 2nd through 31st bytes \*)

DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);

DEST[MAX\_VL-1:256] ← 0;

**VPSUBUSB (VEX.128 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);

(\* Repeat subtract operation for 2nd through 14th bytes \*)

DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);

DEST[MAX\_VL-1:128] ← 0

**PSUBUSB (128-bit Legacy SSE Version)**

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);

(\* Repeat subtract operation for 2nd through 14th bytes \*)

DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);

DEST[MAX\_VL-1:128] (Unmodified)

**VPSUBUSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);

(\* Repeat subtract operation for 2nd through 15th words \*)

DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);

DEST[MAX\_VL-1:256] ← 0;

**VPSUBUSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);

(\* Repeat subtract operation for 2nd through 7th words \*)

DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);

DEST[MAX\_VL-1:128] ← 0

**PSUBUSW (128-bit Legacy SSE Version)**

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);

(\* Repeat subtract operation for 2nd through 7th words \*)

DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSUBUSB \_\_m512i \_mm512\_subs\_epu8(\_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m512i \_mm512\_mask\_subs\_epu8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m512i \_mm512\_maskz\_subs\_epu8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m256i \_mm256\_mask\_subs\_epu8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSB \_\_m256i \_mm256\_maskz\_subs\_epu8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSB \_\_m128i \_mm\_mask\_subs\_epu8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSB \_\_m128i \_mm\_maskz\_subs\_epu8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSW \_\_m512i \_mm512\_subs\_epu16(\_\_m512i a, \_\_m512i b);  
 VPSUBUSW \_\_m512i \_mm512\_mask\_subs\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSW \_\_m512i \_mm512\_maskz\_subs\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSW \_\_m256i \_mm256\_mask\_subs\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSW \_\_m256i \_mm256\_maskz\_subs\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSW \_\_m128i \_mm\_mask\_subs\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSW \_\_m128i \_mm\_maskz\_subs\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBUSB \_\_m128i \_mm\_subs\_epu8(\_\_m128i m1, \_\_m128i m2)  
 PSUBUSW \_\_m128i \_mm\_subs\_epu16(\_\_m128i m1, \_\_m128i m2)  
 PSUBUSB \_\_m256i \_mm256\_subs\_epu8(\_\_m256i m1, \_\_m256i m2)  
 PSUBUSW \_\_m256i \_mm256\_subs\_epu16(\_\_m256i m1, \_\_m256i m2)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## VPTESTNMB/W/D/Q—Logical NAND and Set

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID                | Description   |
|--|-----------|------------------------------|----------------------|---|
| EVEX.NDS.128.F3.0F38.W0 26 /r<br>VPTESTNMB k2 {k1}, xmm2,<br>xmm3/m128         | FVM       | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.256.F3.0F38.W0 26 /r<br>VPTESTNMB k2 {k1}, ymm2,<br>ymm3/m256         | FVM       | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.512.F3.0F38.W0 26 /r<br>VPTESTNMB k2 {k1}, zmm2,<br>zmm3/m512         | FVM       | V/V                          | AVX512F<br>AVX512BW  | Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.128.F3.0F38.W1 26 /r<br>VPTESTNMW k2 {k1}, xmm2,<br>xmm3/m128         | FVM       | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.256.F3.0F38.W1 26 /r<br>VPTESTNMW k2 {k1}, ymm2,<br>ymm3/m256         | FVM       | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.512.F3.0F38.W1 26 /r<br>VPTESTNMW k2 {k1}, zmm2,<br>zmm3/m512         | FVM       | V/V                          | AVX512F<br>AVX512BW  | Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.128.F3.0F38.W0 27 /r<br>VPTESTNMD k2 {k1}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1. |
| EVEX.NDS.256.F3.0F38.W0 27 /r<br>VPTESTNMD k2 {k1}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1. |
| EVEX.NDS.512.F3.0F38.W0 27 /r<br>VPTESTNMD k2 {k1}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F              | Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1. |
| EVEX.NDS.128.F3.0F38.W1 27 /r<br>VPTESTNMQ k2 {k1}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.   |
| EVEX.NDS.256.F3.0F38.W1 27 /r<br>VPTESTNMQ k2 {k1}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.   |
| EVEX.NDS.512.F3.0F38.W1 27 /r<br>VPTESTNMQ k2 {k1}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F              | Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.   |

## Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FVM   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

## Operation

**VPTESTNMB**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

    i ← j\*8

    IF MaskBit(j) OR \*no writemask\*

        THEN

            DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0

        ELSE DEST[j] ← 0; zeroing masking only

    FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMW**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

    i ← j\*16

    IF MaskBit(j) OR \*no writemask\*

        THEN

            DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0

        ELSE DEST[j] ← 0; zeroing masking only

    FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j\*32

    IF MaskBit(j) OR \*no writemask\*

        THEN

            IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

                THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0

            ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0

        FI

```

        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VPTESTNMQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPTESTNMB __mmask64 __mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTESTNMB __mmask64 __mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 __mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 __mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 __mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 __mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 __mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTESTNMW __mmask32 __mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 __mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 __mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 __mm_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 __mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 __mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTESTNMD __mmask16 __mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 __mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 __mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 __mm_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 __mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 __mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm_testn_epi64_mask(__m128i a, __m128i b);
VPTESTNMQ __mmask8 __mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPTESTNMD/VPTESTNMQ: See Exceptions Type E4.

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ—Unpack High Data

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| 66 0F 68 /r<br>PUNPCKHBW xmm1,xmm2/m128  | RM         | V/V                          | SSE2                     | Interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.  |
| 66 0F 69 /r<br>PUNPCKHWD xmm1,xmm2/m128  | RM         | V/V                          | SSE2                     | Interleave high-order words from xmm1 and xmm2/m128 into xmm1.  |
| 66 0F 6A /r<br>PUNPCKHDQ xmm1, xmm2/m128   | RM         | V/V                          | SSE2                     | Interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.                                      |
| 66 0F 6D /r<br>PUNPCKHQDQ xmm1, xmm2/m128  | RM         | V/V                          | SSE2                     | Interleave high-order quadword from xmm1 and xmm2/m128 into xmm1 register.                                |
| VEX.NDS.128.66.0F.WIG 68 /r<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>VPUNPCKHQDQ ymm1, ymm2, ymm3/m256                   | RVM        | V/V                          | AVX2                     | Interleave high-order quadword from ymm2 and ymm3/m256 into ymm1 register.                                |
| EVEX.NDS.128.66.0F.WIG 68 /r<br>VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128           | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.               |
| EVEX.NDS.128.66.0F.WIG 69 /r<br>VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128           | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.               |
| EVEX.NDS.128.66.0F.W0 6A /r<br>VPUNPCKHDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask. |

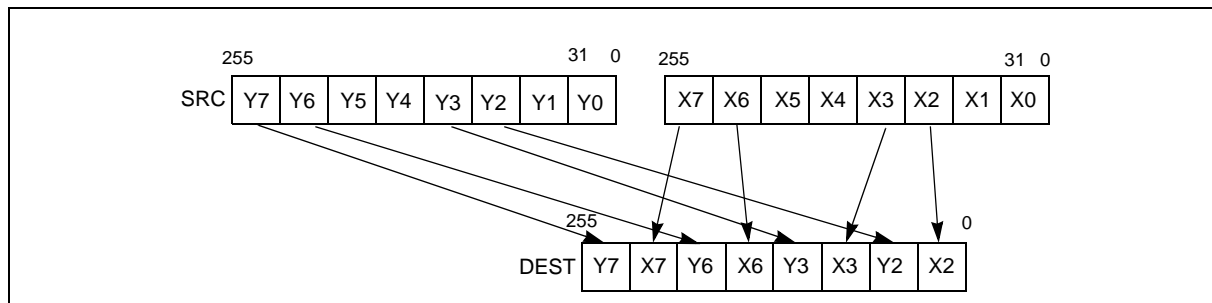
| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F.W1 6D /r<br>VPUNPCKHQDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave high-order quadword from<br>xmm2 and xmm3/m128/m64bcst into<br>xmm1 register using k1 write mask.    |
| EVEX.NDS.256.66.0F.WIG 68 /r<br>VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256            | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave high-order bytes from ymm2 and<br>ymm3/m256 into ymm1 register using k1<br>write mask.               |
| EVEX.NDS.256.66.0F.WIG 69 /r<br>VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256            | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave high-order words from ymm2<br>and ymm3/m256 into ymm1 register using<br>k1 write mask.               |
| EVEX.NDS.256.66.0F.W0 6A /r<br>VPUNPCKHDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst  | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave high-order doublewords from<br>ymm2 and ymm3/m256/m32bcst into<br>ymm1 register using k1 write mask. |
| EVEX.NDS.256.66.0F.W1 6D /r<br>VPUNPCKHQDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave high-order quadword from<br>ymm2 and ymm3/m256/m64bcst into<br>ymm1 register using k1 write mask.    |
| EVEX.NDS.512.66.0F 68/r<br>VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512                 | FVM        | V/V                          | AVX512BW                 | Interleave high-order bytes from zmm2 and<br>zmm3/m512 into zmm1 register.                                      |
| EVEX.NDS.512.66.0F 69/r<br>VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512                 | FVM        | V/V                          | AVX512BW                 | Interleave high-order words from zmm2<br>and zmm3/m512 into zmm1 register.                                      |
| EVEX.NDS.512.66.0F.W0 6A /r<br>VPUNPCKHDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst  | FV         | V/V                          | AVX512F                  | Interleave high-order doublewords from<br>zmm2 and zmm3/m512/m32bcst into<br>zmm1 register using k1 write mask. |
| EVEX.NDS.512.66.0F.W1 6D /r<br>VPUNPCKHQDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Interleave high-order quadword from zmm2<br>and zmm3/m512/m64bcst into zmm1<br>register using k1 write mask.    |

### 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

#### Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-35 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.



**Figure 5-35. 256-bit VPUNPCKHDQ Instruction Operation**

When the source data comes from a memory operand, an implementation may fetch only the appropriate half of the bits (e.g., 64 bits in 128-bit case); however, alignment rules and normal segment checking will still be enforced.

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

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 (MAX\_VL-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 (MAX\_VL-1:128) of the corresponding destination 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.

EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

### Operation

INTERLEAVE\_HIGH\_BYTES\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[255:0], SRC[255:0])

TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_HIGH\_BYTES\_256b (SRC1, SRC2)

DEST[7:0] ← SRC1[71:64]

DEST[15:8] ← SRC2[71:64]

DEST[23:16] ← SRC1[79:72]

DEST[31:24] ← SRC2[79:72]

DEST[39:32] ← SRC1[87:80]

DEST[47:40] ← SRC2[87:80]

DEST[55:48] ← SRC1[95:88]

DEST[63:56] ← SRC2[95:88]

DEST[71:64] ← SRC1[103:96]

DEST[79:72] ← SRC2[103:96]

DEST[87:80] ← SRC1[111:104]

DEST[95:88] ← SRC2[111:104]



DEST[103:96] ← SRC1[119:112]  
 DEST[111:104] ← SRC2[119:112]  
 DEST[119:112] ← SRC1[127:120]  
 DEST[127:120] ← SRC2[127:120]  
 DEST[135:128] ← SRC1[199:192]  
 DEST[143:136] ← SRC2[199:192]  
 DEST[151:144] ← SRC1[207:200]  
 DEST[159:152] ← SRC2[207:200]  
 DEST[167:160] ← SRC1[215:208]  
 DEST[175:168] ← SRC2[215:208]  
 DEST[183:176] ← SRC1[223:216]  
 DEST[191:184] ← SRC2[223:216]  
 DEST[199:192] ← SRC1[231:224]  
 DEST[207:200] ← SRC2[231:224]  
 DEST[215:208] ← SRC1[239:232]  
 DEST[223:216] ← SRC2[239:232]  
 DEST[231:224] ← SRC1[247:240]  
 DEST[239:232] ← SRC2[247:240]  
 DEST[247:240] ← SRC1[255:248]  
 DEST[255:248] ← SRC2[255:248]

INTERLEAVE\_HIGH\_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[71:64]  
 DEST[15:8] ← SRC2[71:64]  
 DEST[23:16] ← SRC1[79:72]  
 DEST[31:24] ← SRC2[79:72]  
 DEST[39:32] ← SRC1[87:80]  
 DEST[47:40] ← SRC2[87:80]  
 DEST[55:48] ← SRC1[95:88]  
 DEST[63:56] ← SRC2[95:88]  
 DEST[71:64] ← SRC1[103:96]  
 DEST[79:72] ← SRC2[103:96]  
 DEST[87:80] ← SRC1[111:104]  
 DEST[95:88] ← SRC2[111:104]  
 DEST[103:96] ← SRC1[119:112]  
 DEST[111:104] ← SRC2[119:112]  
 DEST[119:112] ← SRC1[127:120]  
 DEST[127:120] ← SRC2[127:120]

INTERLEAVE\_HIGH\_WORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[255:0], SRC[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]  
 DEST[31:16] ← SRC2[79:64]  
 DEST[47:32] ← SRC1[95:80]  
 DEST[63:48] ← SRC2[95:80]  
 DEST[79:64] ← SRC1[111:96]  
 DEST[95:80] ← SRC2[111:96]  
 DEST[111:96] ← SRC1[127:112]  
 DEST[127:112] ← SRC2[127:112]  
 DEST[143:128] ← SRC1[207:192]  
 DEST[159:144] ← SRC2[207:192]

DEST[175:160] ← SRC1[223:208]  
 DEST[191:176] ← SRC2[223:208]  
 DEST[207:192] ← SRC1[239:224]  
 DEST[223:208] ← SRC2[239:224]  
 DEST[239:224] ← SRC1[255:240]  
 DEST[255:240] ← SRC2[255:240]

INTERLEAVE\_HIGH\_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]  
 DEST[31:16] ← SRC2[79:64]  
 DEST[47:32] ← SRC1[95:80]  
 DEST[63:48] ← SRC2[95:80]  
 DEST[79:64] ← SRC1[111:96]  
 DEST[95:80] ← SRC2[111:96]  
 DEST[111:96] ← SRC1[127:112]  
 DEST[127:112] ← SRC2[127:112]

INTERLEAVE\_HIGH\_DWORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1, SRC2)

DEST[31:0] ← SRC1[95:64]  
 DEST[63:32] ← SRC2[95:64]  
 DEST[95:64] ← SRC1[127:96]  
 DEST[127:96] ← SRC2[127:96]  
 DEST[159:128] ← SRC1[223:192]  
 DEST[191:160] ← SRC2[223:192]  
 DEST[223:192] ← SRC1[255:224]  
 DEST[255:224] ← SRC2[255:224]

INTERLEAVE\_HIGH\_DWORDS(SRC1, SRC2)

DEST[31:0] ← SRC1[95:64]  
 DEST[63:32] ← SRC2[95:64]  
 DEST[95:64] ← SRC1[127:96]  
 DEST[127:96] ← SRC2[127:96]

INTERLEAVE\_HIGH\_QWORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1, SRC2)

DEST[63:0] ← SRC1[127:64]  
 DEST[127:64] ← SRC2[127:64]  
 DEST[191:128] ← SRC1[255:192]  
 DEST[255:192] ← SRC2[255:192]

INTERLEAVE\_HIGH\_QWORDS(SRC1, SRC2)

DEST[63:0] ← SRC1[127:64]  
 DEST[127:64] ← SRC2[127:64]

#### **PUNPCKHBW (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_BYTES(DEST, SRC)  
 DEST[255:127] (Unmodified)

**VPUNPCKHBW (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_BYTES(SRC1, SRC2)

DEST[511:127] ← 0

**VPUNPCKHBW (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPUNPCKHBW (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**PUNPCKHWD (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKHWD (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_WORDS(SRC1, SRC2)

DEST[511:127] ← 0

**VPUNPCKHWD (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPUNPCKHWD (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

```
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[j+15:i] ← TMP_DEST[j+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[j+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

**PUNPCKHDQ (128-bit Legacy SSE Version)**

```
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

**VPUNPCKHDQ (VEX.128 encoded version)**

```
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[511:127] ← 0
```

**VPUNPCKHDQ (VEX.256 encoded version)**

```
DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0
```

**VPUNPCKHDQ (EVEX.512 encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[j+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[j+31:i] ← SRC2[j+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
    ELSE
```

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**PUNPCKHQDQ (128-bit Legacy SSE Version)**

```

DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**VPUNPCKHQDQ (VEX.128 encoded version)**

```

DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPUNPCKHQDQ (VEX.256 encoded version)**

```

DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0

```

**VPUNPCKHQDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

```

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*     ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPUNPCKHBW \_\_m512i \_mm512\_unpackhi\_epi8(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHBW \_\_m512i \_mm512\_mask\_unpackhi\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHBW \_\_m512i \_mm512\_maskz\_unpackhi\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHBW \_\_m256i \_mm256\_mask\_unpackhi\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHBW \_\_m256i \_mm256\_maskz\_unpackhi\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHBW \_\_m128i \_mm\_mask\_unpackhi\_epi8(v s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHBW \_\_m128i \_mm\_maskz\_unpackhi\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHWD \_\_m512i \_mm512\_unpackhi\_epi16(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m512i \_mm512\_mask\_unpackhi\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m512i \_mm512\_maskz\_unpackhi\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m256i \_mm256\_mask\_unpackhi\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHWD \_\_m256i \_mm256\_maskz\_unpackhi\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHWD \_\_m128i \_mm\_mask\_unpackhi\_epi16(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHWD \_\_m128i \_mm\_maskz\_unpackhi\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHDQ \_\_m512i \_mm512\_unpackhi\_epi32(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m512i \_mm512\_mask\_unpackhi\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m512i \_mm512\_maskz\_unpackhi\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m256i \_mm256\_mask\_unpackhi\_epi32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m256i \_mm256\_maskz\_unpackhi\_epi32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m128i \_mm\_mask\_unpackhi\_epi32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m128i \_mm\_maskz\_unpackhi\_epi32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_mm512\_unpackhi\_epi64(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_mm512\_mask\_unpackhi\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_mm512\_maskz\_unpackhi\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m256i \_mm256\_mask\_unpackhi\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m256i \_mm256\_maskz\_unpackhi\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m128i \_mm\_mask\_unpackhi\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m128i \_mm\_maskz\_unpackhi\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 (V)PUNPCKHBW \_\_m128i \_mm\_unpackhi\_epi8(\_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHBW \_\_m256i \_mm256\_unpackhi\_epi8(\_\_m256i m1, \_\_m256i m2)  
 (V)PUNPCKHWD \_\_m128i \_mm\_unpackhi\_epi16(\_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHWD \_\_m256i \_mm256\_unpackhi\_epi16(\_\_m256i m1, \_\_m256i 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)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.

**PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data**

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| 66 0F 60 /r<br>PUNPCKLBW xmm1,xmm2/m128  | RM         | V/V                          | SSE2                     | Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.   |
| 66 0F 61 /r<br>PUNPCKLWD xmm1,xmm2/m128  | RM         | V/V                          | SSE2                     | Interleave low-order words from xmm1 and xmm2/m128 into xmm1.   |
| 66 0F 62 /r<br>PUNPCKLDQ xmm1, xmm2/m128   | RM         | V/V                          | SSE2                     | Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.   |
| 66 0F 6C /r<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>VPUNPCKLQDQ ymm1, ymm2, ymm3/m256                   | RVM        | V/V                          | AVX2                     | Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.                                     |
| EVEX.NDS.128.66.0F.WIG 60 /r<br>VPUNPCKLBW xmm1 {k1}{z}, xmm2,<br>xmm3/m128        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.               |
| EVEX.NDS.128.66.0F.WIG 61 /r<br>VPUNPCKLWD xmm1 {k1}{z}, xmm2,<br>xmm3/m128        | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.               |
| EVEX.NDS.128.66.0F.W0 62 /r<br>VPUNPCKLDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1. |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F.W1 6C /r<br>VPUNPCKLQDQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.    |
| EVEX.NDS.256.66.0F.WIG 60 /r<br>VPUNPCKLBW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.               |
| EVEX.NDS.256.66.0F.WIG 61 /r<br>VPUNPCKLWD ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | FVM        | V/V                          | AVX512VL<br>AVX512BW     | Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.               |
| EVEX.NDS.256.66.0F.W0 62 /r<br>VPUNPCKLDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst  | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1. |
| EVEX.NDS.256.66.0F.W1 6C /r<br>VPUNPCKLQDQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.    |
| EVEX.NDS.512.66.0F 60/r<br>VPUNPCKLBW zmm1 {k1}{z}, zmm2,<br>zmm3/m512              | FVM        | V/V                          | AVX512BW                 | Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.               |
| EVEX.NDS.512.66.0F 61/r<br>VPUNPCKLWD zmm1 {k1}{z}, zmm2,<br>zmm3/m512              | FVM        | V/V                          | AVX512BW                 | Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.               |
| EVEX.NDS.512.66.0F.W0 62 /r<br>VPUNPCKLDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst  | FV         | V/V                          | AVX512F                  | Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1. |
| EVEX.NDS.512.66.0F.W1 6C /r<br>VPUNPCKLQDQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.    |

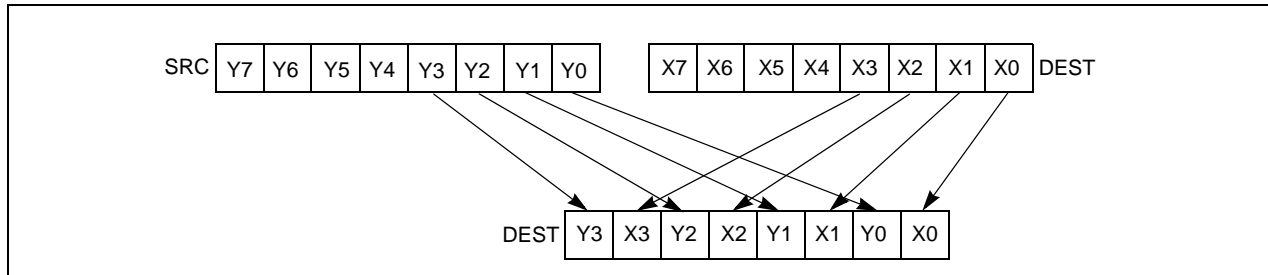
### Instruction Operand Encoding

| 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        |
| FVM   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-36 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.





**Figure 5-36. 128-bit PUNPCKLBW Instruction Operation using 64-bit Operands**

When the source data comes from a memory operand, an implementation may fetch only the appropriate half of the bits (e.g., 64 bits in 128-bit case); however, alignment rules and normal segment checking will still be enforced.

The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the PUNPCKLDQ instruction interleaves the low order doubleword (or doublewords) of the source and destination operands, and the PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

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).

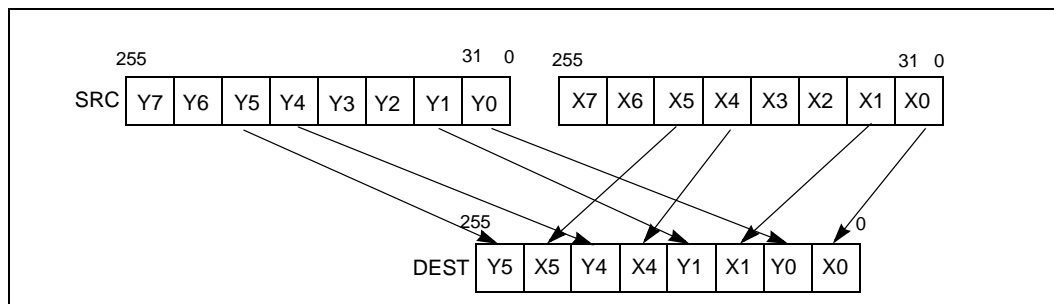
128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

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 (MAX\_VL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The second source operand is a YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.



**Figure 5-37. 256-bit VPUNPCKLDQ Instruction Operation**

**Operation**

INTERLEAVE\_BYTES\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_BYTES\_256b(SRC1[255:0], SRC[255:0])

TMP\_DEST[511:256] ← INTERLEAVE\_BYTES\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_BYTES\_256b (SRC1, SRC2)

DEST[7:0] ← SRC1[7:0]

DEST[15:8] ← SRC2[7:0]

DEST[23:16] ← SRC1[15:8]

DEST[31:24] ← SRC2[15:8]

DEST[39:32] ← SRC1[23:16]

DEST[47:40] ← SRC2[23:16]

DEST[55:48] ← SRC1[31:24]

DEST[63:56] ← SRC2[31:24]

DEST[71:64] ← SRC1[39:32]

DEST[79:72] ← SRC2[39:32]

DEST[87:80] ← SRC1[47:40]

DEST[95:88] ← SRC2[47:40]

DEST[103:96] ← SRC1[55:48]

DEST[111:104] ← SRC2[55:48]

DEST[119:112] ← SRC1[63:56]

DEST[127:120] ← SRC2[63:56]

DEST[135:128] ← SRC1[135:128]

DEST[143:136] ← SRC2[135:128]

DEST[151:144] ← SRC1[143:136]

DEST[159:152] ← SRC2[143:136]

DEST[167:160] ← SRC1[151:144]

DEST[175:168] ← SRC2[151:144]

DEST[183:176] ← SRC1[159:152]

DEST[191:184] ← SRC2[159:152]

DEST[199:192] ← SRC1[167:160]

DEST[207:200] ← SRC2[167:160]

DEST[215:208] ← SRC1[175:168]

DEST[223:216] ← SRC2[175:168]

DEST[231:224] ← SRC1[183:176]

DEST[239:232] ← SRC2[183:176]

DEST[247:240] ← SRC1[191:184]

DEST[255:248] ← SRC2[191:184]

INTERLEAVE\_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[7:0]

DEST[15:8] ← SRC2[7:0]

DEST[23:16] ← SRC2[15:8]

DEST[31:24] ← SRC2[15:8]

DEST[39:32] ← SRC1[23:16]

DEST[47:40] ← SRC2[23:16]

DEST[55:48] ← SRC1[31:24]

DEST[63:56] ← SRC2[31:24]

DEST[71:64] ← SRC1[39:32]

DEST[79:72] ← SRC2[39:32]

DEST[87:80] ← SRC1[47:40]

DEST[95:88] ← SRC2[47:40]

DEST[103:96] ← SRC1[55:48]

DEST[111:104] ← SRC2[55:48]

DEST[119:112] ← SRC1[63:56]  
 DEST[127:120] ← SRC2[63:56]

INTERLEAVE\_WORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_WORDS\_256b(SRC1[255:0], SRC[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_WORDS\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_WORDS\_256b(SRC1, SRC2)  
 DEST[15:0] ← SRC1[15:0]  
 DEST[31:16] ← SRC2[15:0]  
 DEST[47:32] ← SRC1[31:16]  
 DEST[63:48] ← SRC2[31:16]  
 DEST[79:64] ← SRC1[47:32]  
 DEST[95:80] ← SRC2[47:32]  
 DEST[111:96] ← SRC1[63:48]  
 DEST[127:112] ← SRC2[63:48]  
 DEST[143:128] ← SRC1[143:128]  
 DEST[159:144] ← SRC2[143:128]  
 DEST[175:160] ← SRC1[159:144]  
 DEST[191:176] ← SRC2[159:144]  
 DEST[207:192] ← SRC1[175:160]  
 DEST[223:208] ← SRC2[175:160]  
 DEST[239:224] ← SRC1[191:176]  
 DEST[255:240] ← SRC2[191:176]

INTERLEAVE\_WORDS (SRC1, SRC2)  
 DEST[15:0] ← SRC1[15:0]  
 DEST[31:16] ← SRC2[15:0]  
 DEST[47:32] ← SRC1[31:16]  
 DEST[63:48] ← SRC2[31:16]  
 DEST[79:64] ← SRC1[47:32]  
 DEST[95:80] ← SRC2[47:32]  
 DEST[111:96] ← SRC1[63:48]  
 DEST[127:112] ← SRC2[63:48]

INTERLEAVE\_DWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_DWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_DWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_DWORDS\_256b(SRC1, SRC2)  
 DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[159:128] ← SRC1[159:128]  
 DEST[191:160] ← SRC2[159:128]  
 DEST[223:192] ← SRC1[191:160]  
 DEST[255:224] ← SRC2[191:160]

INTERLEAVE\_DWORDS(SRC1, SRC2)  
 DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]

INTERLEAVE\_QWORDS\_512b(SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_QWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_QWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_QWORDS\_256b(SRC1, SRC2)  
 DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[191:128] ← SRC1[191:128]  
 DEST[255:192] ← SRC2[191:128]

INTERLEAVE\_QWORDS(SRC1, SRC2)  
 DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]

**PUNPCKLBW**

DEST[127:0] ← INTERLEAVE\_BYTES(DEST, SRC)  
 DEST[255:127] (Unmodified)

**VPUNPCKLBW (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_BYTES(SRC1, SRC2)  
 DEST[511:127] ← 0

**VPUNPCKLBW (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_BYTES\_256b(SRC1, SRC2)  
 DEST[MAX\_VL-1:256] ← 0

**VPUNPCKLBW (EVEX.512 encoded instruction)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

    TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

    TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

    TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

    i ← j \* 8

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+7:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+7:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

DEST[511:0] ← INTERLEAVE\_BYTES\_512b(SRC1, SRC2)

**PUNPCKLWD**

DEST[127:0] ← INTERLEAVE\_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKLWD (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_WORDS(SRC1, SRC2)

DEST[511:127] ← 0

**VPUNPCKLWD (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPUNPCKLWD (EVEX.512 encoded instruction)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

DEST[511:0] ← INTERLEAVE\_WORDS\_512b(SRC1, SRC2)

**PUNPCKLDQ**

DEST[127:0] ← INTERLEAVE\_DWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**VPUNPCKLDQ (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_DWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPUNPCKLDQ (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_DWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPUNPCKLDQ (EVEX encoded instructions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[511:0] ← INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAX_VL-1:VL] ← 0

```

**PUNPCKLDQ**

DEST[127:0] ← INTERLEAVE\_QWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**VPUNPCKLDQ (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_QWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPUNPCKLDQ (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_QWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPUNPCKLDQ (EVEX encoded instructions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;

```

```

ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b);

```

(V)PUNPCKLBW \_\_m128i \_\_mm\_unpacklo\_epi8 (\_\_m128i m1, \_\_m128i m2)  
VPUNPCKLBW \_\_m256i \_\_mm256\_unpacklo\_epi8 (\_\_m256i m1, \_\_m256i m2)  
(V)PUNPCKLWD \_\_m128i \_\_mm\_unpacklo\_epi16 (\_\_m128i m1, \_\_m128i m2)  
VPUNPCKLWD \_\_m256i \_\_mm256\_unpacklo\_epi16 (\_\_m256i m1, \_\_m256i 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)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKLDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.



## SHUFF32x4/SHUFF64x2/SHUFI32x4/SHUFI64x2—Shuffle Packed Values at 128-bit Granularity

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.256.66.0F3A.W0 23 /r ib<br>VSHUFF32X4 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F3A.W0 23 /r ib<br>VSHUFF32x4 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | FV        | V/V                          | AVX512F                  | Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F3A.W1 23 /r ib<br>VSHUFF64X2 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1. |
| EVEX.NDS.512.66.0F3A.W1 23 /r ib<br>VSHUFF64x2 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | FV        | V/V                          | AVX512F                  | Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1. |
| EVEX.NDS.256.66.0F3A.W0 43 /r ib<br>VSHUFI32X4 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.                     |
| EVEX.NDS.512.66.0F3A.W0 43 /r ib<br>VSHUFI32x4 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | FV        | V/V                          | AVX512F                  | Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.                     |
| EVEX.NDS.256.66.0F3A.W1 43 /r ib<br>VSHUFI64X2 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.                       |
| EVEX.NDS.512.66.0F3A.W1 43 /r ib<br>VSHUFI64x2 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | FV        | V/V                          | AVX512F                  | Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.                       |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

**256-bit Version:** Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

**512-bit Version:** Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

### Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
  0:  TMP ← SRC[127:0];
  1:  TMP ← SRC[255:128];
ESAC;
RETURN TMP
}
```

```
Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP ← SRC[127:0];
  1:  TMP ← SRC[255:128];
  2:  TMP ← SRC[383:256];
  3:  TMP ← SRC[511:384];
ESAC;
RETURN TMP
}
```

### VSHUFF32x4 (EVEX versions)

(KL, VL) = (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          THEN DEST[i+31:i] ← 0
      FI;
    FI;
  FI;
ENDFOR;
```

```

ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VSHUFF64x2 (EVEX 512-bit version)

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          THEN DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VSHUFI32x4 (EVEX 512-bit version)

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);

```

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      THEN DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSHUFI64x2 (EVEX 512-bit version)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[j+63:i] ← SRC2[63:0]
  ELSE TMP_SRC2[j+63:i] ← SRC2[j+63:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      THEN DEST[j+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFI32x4 __m512i __mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i __mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i __mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i __mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i __mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);

```

VSHUFI32x4 \_\_m256i \_\_mm256\_maskz\_shuffle\_i32x4(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VSHUFF32x4 \_\_m512 \_\_mm512\_shuffle\_f32x4(\_\_m512 a, \_\_m512 b, int imm);  
 VSHUFF32x4 \_\_m512 \_\_mm512\_mask\_shuffle\_f32x4(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int imm);  
 VSHUFF32x4 \_\_m512 \_\_mm512\_maskz\_shuffle\_f32x4(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int imm);  
 VSHUFI64x2 \_\_m512i \_\_mm512\_shuffle\_i64x2(\_\_m512i a, \_\_m512i b, int imm);  
 VSHUFI64x2 \_\_m512i \_\_mm512\_mask\_shuffle\_i64x2(\_\_m512i s, \_\_mmask8 k, \_\_m512i b, \_\_m512i b, int imm);  
 VSHUFI64x2 \_\_m512i \_\_mm512\_maskz\_shuffle\_i64x2(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, int imm);  
 VSHUFF64x2 \_\_m512d \_\_mm512\_shuffle\_f64x2(\_\_m512d a, \_\_m512d b, int imm);  
 VSHUFF64x2 \_\_m512d \_\_mm512\_mask\_shuffle\_f64x2(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int imm);  
 VSHUFF64x2 \_\_m512d \_\_mm512\_maskz\_shuffle\_f64x2(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int imm);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4NF.

#UD If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.

## SHUFPD—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F C6 /r ib<br>SHUFPD xmm1, xmm2/m128, imm8  | RMI        | V/V                          | SSE2                     | Shuffle two pairs of double-precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.                                |
| VEX.NDS.128.66.0F.WIG C6 /r ib<br>VSHUFPD xmm1, xmm2, xmm3/m128,<br>imm8                | RVMI       | V/V                          | AVX                      | Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.                                |
| VEX.NDS.256.66.0F.WIG C6 /r ib<br>VSHUFPD ymm1, ymm2, ymm3/m256,<br>imm8                | RVMI       | V/V                          | AVX                      | Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.                               |
| EVEX.NDS.128.0F.W1 C6 /r ib<br>VSHUFPD xmm1{k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8    | FV         | V/V                          | AVX512VL<br>AVX512F      | Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.   |
| EVEX.NDS.256.0F.W1 C6 /r ib<br>VSHUFPD ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8    | FV         | V/V                          | AVX512VL<br>AVX512F      | Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.  |
| EVEX.NDS.512.66.0F.W1 C6 /r ib<br>VSHUFPD zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | FV         | V/V                          | AVX512F                  | Shuffle eight pairs of double-precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

Selects a double-precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double-precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4] are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2] are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

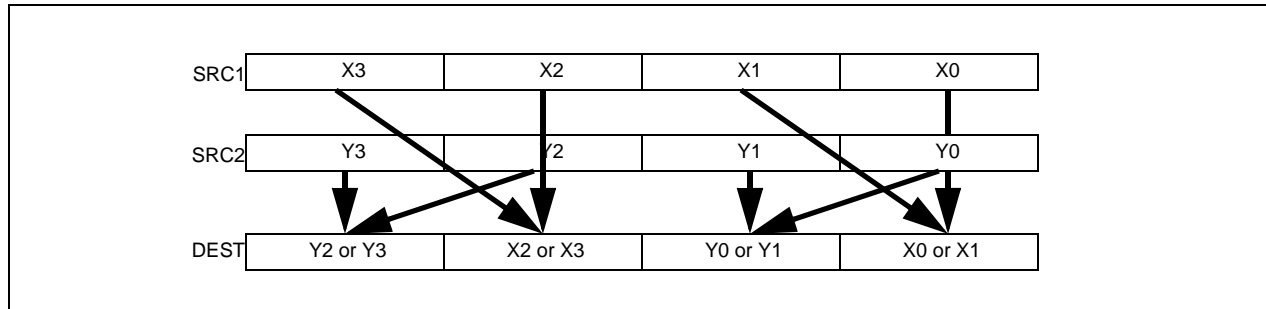


Figure 5-38. 256-bit VSHUFPD Operation of Four Pairs of DP FP Values

### Operation

#### VSHUFPD (EVEX encoded versions when SRC2 is a vector register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF IMMO[0] = 0

THEN TMP\_DEST[63:0] ← SRC1[63:0]

ELSE TMP\_DEST[63:0] ← SRC1[127:64] FI;

IF IMMO[1] = 0

THEN TMP\_DEST[127:64] ← SRC2[63:0]

ELSE TMP\_DEST[127:64] ← SRC2[127:64] FI;

IF VL ≥ 256

IF IMMO[2] = 0

THEN TMP\_DEST[191:128] ← SRC1[191:128]

ELSE TMP\_DEST[191:128] ← SRC1[255:192] FI;

IF IMMO[3] = 0

THEN TMP\_DEST[255:192] ← SRC2[191:128]

ELSE TMP\_DEST[255:192] ← SRC2[255:192] FI;

FI;

IF VL ≥ 512

IF IMMO[4] = 0

THEN TMP\_DEST[319:256] ← SRC1[319:256]

ELSE TMP\_DEST[319:256] ← SRC1[383:320] FI;

IF IMMO[5] = 0

THEN TMP\_DEST[383:320] ← SRC2[319:256]

ELSE TMP\_DEST[383:320] ← SRC2[383:320] FI;

IF IMMO[6] = 0

THEN TMP\_DEST[447:384] ← SRC1[447:384]

ELSE TMP\_DEST[447:384] ← SRC1[511:448] FI;

IF IMMO[7] = 0

THEN TMP\_DEST[511:448] ← SRC2[447:384]

ELSE TMP\_DEST[511:448] ← SRC2[511:448] FI;

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+63:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[j+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSHUFPD (EVEX encoded versions when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[j+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[j+63:i] ← SRC2[j+63:i]
    FI;
ENDFOR;
IF IMMO[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN TMP_DEST[127:64] ← TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] ← TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMMO[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMMO[3] = 0
        THEN TMP_DEST[255:192] ← TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] ← TMP_SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMMO[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMMO[5] = 0
        THEN TMP_DEST[383:320] ← TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] ← TMP_SRC2[383:320] FI;
    IF IMMO[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMMO[7] = 0
        THEN TMP_DEST[511:448] ← TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] ← TMP_SRC2[511:448] FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+63:i] remains unchanged*

```



```

        ELSE *zeroing-masking*           ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSHUFPD (VEX.256 encoded version)**

```

IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMMO[2] = 0
    THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMMO[3] = 0
    THEN DEST[255:192] ← SRC2[191:128]
    ELSE DEST[255:192] ← SRC2[255:192] FI;
DEST[MAX_VL-1:256] (Unmodified)

```

**VSHUFPD (VEX.128 encoded version)**

```

IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[MAX_VL-1:128] ← 0

```

**VSHUFPD (128-bit Legacy SSE version)**

```

IF IMMO[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFPD __m512d __mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d __mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d __mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d __mm256_shuffle_pd(__m256d a, __m256d b, const int select);
VSHUFPD __m256d __mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d __mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d __mm_shuffle_pd(__m128d a, __m128d b, const int select);
VSHUFPD __m128d __mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d __mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

## SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| OF C6 /r ib<br>SHUFPS xmm1, xmm3/m128, imm8  | RMI        | V/V                          | SSE                      | Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.                          |
| VEX.NDS.128.OF.WIG C6 /r ib<br>VSHUFPS xmm1, xmm2,<br>xmm3/m128, imm8                | RVMI       | V/V                          | AVX                      | Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.                          |
| VEX.NDS.256.OF.WIG C6 /r ib<br>VSHUFPS ymm1, ymm2,<br>ymm3/m256, imm8                | RVMI       | V/V                          | AVX                      | Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.                          |
| EVEX.NDS.128.OF.W0 C6 /r ib<br>VSHUFPS xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8 | FV         | V/V                          | AVX512VL<br>AVX512F      | Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1. |
| EVEX.NDS.256.OF.W0 C6 /r ib<br>VSHUFPS ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | FV         | V/V                          | AVX512VL<br>AVX512F      | Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1. |
| EVEX.NDS.512.OF.W0 C6 /r ib<br>VSHUFPS zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | FV         | V/V                          | AVX512F                  | Select from quadruplet of single-precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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      |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128)

of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

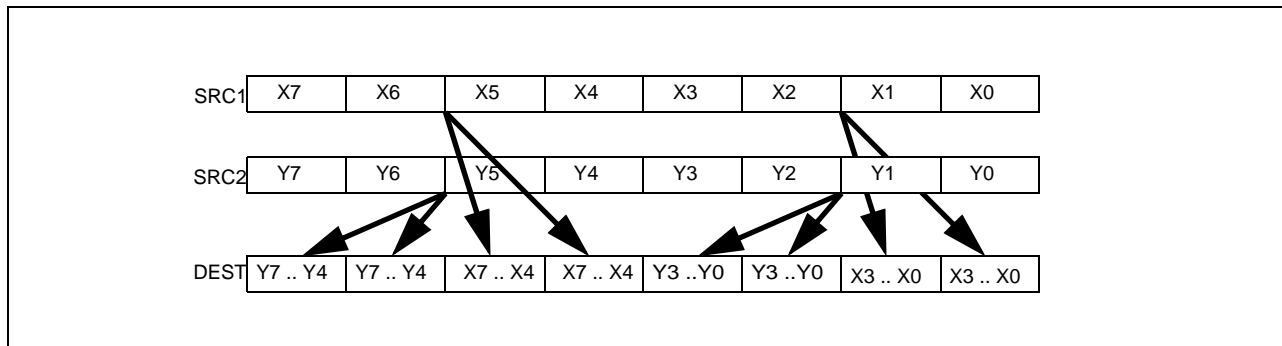


Figure 5-39. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

**Operation**

```
Select4(SRC, control) {
CASE (control[1:0]) OF
0: TMP ← SRC[31:0];
1: TMP ← SRC[63:32];
2: TMP ← SRC[95:64];
3: TMP ← SRC[127:96];
ESAC;
RETURN TMP
}
```

**VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
TMP_DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
TMP_DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
TMP_DEST[351:320] ← Select4(SRC2[383:256], imm8[5:4]);
TMP_DEST[383:352] ← Select4(SRC2[383:256], imm8[7:6]);
TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
TMP_DEST[479:448] ← Select4(SRC2[511:384], imm8[5:4]);
TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
```

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSHUFPS (EVEX encoded versions when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
  ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
  TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
  TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
  TMP_DEST[223:192] ← Select4(TMP_SRC2[255:128], imm8[5:4]);
  TMP_DEST[255:224] ← Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
  TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
  TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
  TMP_DEST[351:320] ← Select4(TMP_SRC2[383:256], imm8[5:4]);
  TMP_DEST[383:352] ← Select4(TMP_SRC2[383:256], imm8[7:6]);
  TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
  TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
  TMP_DEST[479:448] ← Select4(TMP_SRC2[511:384], imm8[5:4]);
  TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

```

    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0
VSHUFPS (VEX.256 encoded version)
  DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
  DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
  DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
  DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
  DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
  DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
  DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
  DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
  DEST[MAX_VL-1:256] ← 0

```

```

VSHUFPS (VEX.128 encoded version)
  DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
  DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
  DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
  DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
  DEST[MAX_VL-1:128] ← 0

```

```

SHUFPS (128-bit Legacy SSE version)
  DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
  DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
  DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
  DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
  DEST[MAX_VL-1:128] (Unmodified)

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VSHUFPS __m512 __mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 __mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 __mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 __mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 __mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 __mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 __mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 __mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 __mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

## SQRTPD—Square Root of Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| 66 0F 51 /r<br>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 result in xmm1.                                 |
| VEX.128.66.0F.WIG 51 /r<br>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<br>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.                                 |
| EVEX.128.0F.W1 51 /r<br>VSQRTPD xmm1 {k1}{z},<br>xmm2/m128/m32bcst        | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes Square Roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1. |
| EVEX.256.0F.W1 51 /r<br>VSQRTPD ymm1 {k1}{z},<br>ymm2/m256/m32bcst        | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes Square Roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1. |
| EVEX.512.66.0F.W1 51 /r<br>VSQRTPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Computes Square Roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Performs a SIMD computation of the square roots of the two, four or eight packed double-precision floating-point values in the source operand (the second operand) stores the packed double-precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation****VSQRTPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC \*is register\*)

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+63:i] ← SQRT(SRC[63:0])
            ELSE DEST[i+63:i] ← SQRT(SRC[i+63:i])
        FI;
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSQRTPD (VEX.256 encoded version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[191:128] ← SQRT(SRC[191:128])

DEST[255:192] ← SQRT(SRC[255:192])

DEST[MAX\_VL-1:256] ← 0

**VSQRTPD (VEX.128 encoded version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[MAX\_VL-1:128] ← 0

**SQRTPD (128-bit Legacy SSE version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTPD \_\_m512d \_\_mm512\_sqrt\_round\_pd(\_\_m512d a, int r);

VSQRTPD \_\_m512d \_\_mm512\_mask\_sqrt\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int r);

VSQRTPD \_\_m512d \_\_mm512\_maskz\_sqrt\_round\_pd(\_\_mmask8 k, \_\_m512d a, int r);

VSQRTPD \_\_m256d \_\_mm256\_sqrt\_pd(\_\_m256d a);

VSQRTPD \_\_m256d \_\_mm256\_mask\_sqrt\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int r);

VSQRTPD \_\_m256d \_\_mm256\_maskz\_sqrt\_pd(\_\_mmask8 k, \_\_m256d a, int r);

SQRTPD \_\_m128d \_\_mm\_sqrt\_pd(\_\_m128d a);

VSQRTPD \_\_m128d \_\_mm\_mask\_sqrt\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int r);

VSQRTPD \_\_m128d \_\_mm\_maskz\_sqrt\_pd(\_\_mmask8 k, \_\_m128d a, int r);



**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## SQRTPS—Square Root of Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| OF 51 /r<br>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 result in xmm1.                                 |
| VEX.128.OF.WIG 51 /r<br>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.OF.WIG 51/r<br>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.                                 |
| EVEX.128.OF.W0 51 /r<br>VSQRTPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst    | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1. |
| EVEX.256.OF.W0 51 /r<br>VSQRTPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst    | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1. |
| EVEX.512.OF.W0 51/r<br>VSQRTPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation****VSQRTPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC \*is register\*)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← SQRT(SRC[31:0])

ELSE DEST[i+31:i] ← SQRT(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VSQRTPS (VEX.256 encoded version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[159:128] ← SQRT(SRC[159:128])

DEST[191:160] ← SQRT(SRC[191:160])

DEST[223:192] ← SQRT(SRC[223:192])

DEST[255:224] ← SQRT(SRC[255:224])

**VSQRTPS (VEX.128 encoded version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[MAX\_VL-1:128] ← 0

**SQRTPS (128-bit Legacy SSE version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTPS __m512 __mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 __mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 __mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 __mm256_sqrt_ps(__m256 a);
VSQRTPS __m256 __mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 __mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 __mm_sqrt_ps(__m128 a);
VSQRTPS __m128 __mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 __mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| F2 0F 51/r<br>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.128.F2.0F.WIG 51/r<br>VSQRTSD xmm1,xmm2,<br>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 xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].                    |
| EVEX.NDS.LIG.F2.0F.W1 51/r<br>VSQRTSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64]. |

### Instruction Operand Encoding

| Op/En | 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        |
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSQRTSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VSQRTSD (VEX.128 encoded version)**

```

DEST[63:0] ← SQRT(SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**SQRTSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SQRT(SRC[63:0])
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTSD __m128d __mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d __mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d __mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d __mm_sqrt_sd (__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## SQRTSS—Compute Square Root of Scalar Single-Precision Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| F3 0F 51 /r<br>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.128.F3.0F.WIG 51 /r<br>VSQRTSS xmm1, xmm2,<br>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].                    |
| EVEX.NDS.LIG.F3.0F.W0 51 /r<br>VSQRTSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S       | V/V                          | AVX512F                  | Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | 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        |
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSQRTSS (EVEX encoded version)**

IF (EVEX.b = 1) AND (SRC2 \*is register\*)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SQRT(SRC2[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

DEST[127:31] ← SRC1[127:31]

DEST[MAX\_VL-1:128] ← 0

**VSQRTSS (VEX.128 encoded version)**

DEST[31:0] ← SQRT(SRC2[31:0])

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

**SQRTSS (128-bit Legacy SSE version)**

DEST[31:0] ← SQRT(SRC2[31:0])

DEST[MAX\_VL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTSS \_\_m128 \_mm\_sqrt\_round\_ss(\_\_m128 a, \_\_m128 b, int r);

VSQRTSS \_\_m128 \_mm\_mask\_sqrt\_round\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int r);

VSQRTSS \_\_m128 \_mm\_maskz\_sqrt\_round\_ss(\_\_mmask8 k, \_\_m128 a, \_\_m128 b, int r);

SQRTSS \_\_m128 \_mm\_sqrt\_ss(\_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.



## VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.DDS.128.66.0F3A.W0 25 /r ib<br>VPTERNLOGD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise ternary logic taking xmm1, xmm2 and<br>xmm3/m128/m32bcst as source operands and writing<br>the result to xmm1 under writemask k1 with dword<br>granularity. The immediate value determines the specific<br>binary function being implemented. |
| EVEX.DDS.256.66.0F3A.W0 25 /r ib<br>VPTERNLOGD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise ternary logic taking ymm1, ymm2 and<br>ymm3/m256/m32bcst as source operands and writing<br>the result to ymm1 under writemask k1 with dword<br>granularity. The immediate value determines the specific<br>binary function being implemented. |
| EVEX.DDS.512.66.0F3A.W0 25 /r ib<br>VPTERNLOGD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | FV        | V/V                          | AVX512F                  | Bitwise ternary logic taking zmm1, zmm2 and<br>zmm3/m512/m32bcst as source operands and writing<br>the result to zmm1 under writemask k1 with dword<br>granularity. The immediate value determines the specific<br>binary function being implemented. |
| EVEX.DDS.128.66.0F3A.W1 25 /r ib<br>VPTERNLOGQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise ternary logic taking xmm1, xmm2 and<br>xmm3/m128/m64bcst as source operands and writing<br>the result to xmm1 under writemask k1 with qword<br>granularity. The immediate value determines the specific<br>binary function being implemented. |
| EVEX.DDS.256.66.0F3A.W1 25 /r ib<br>VPTERNLOGQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise ternary logic taking ymm1, ymm2 and<br>ymm3/m256/m64bcst as source operands and writing<br>the result to ymm1 under writemask k1 with qword<br>granularity. The immediate value determines the specific<br>binary function being implemented. |
| EVEX.DDS.512.66.0F3A.W1 25 /r ib<br>VPTERNLOGQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | FV        | V/V                          | AVX512F                  | Bitwise ternary logic taking zmm1, zmm2 and<br>zmm3/m512/m64bcst as source operands and writing<br>the result to zmm1 under writemask k1 with qword<br>granularity. The immediate value determines the specific<br>binary function being implemented. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------------|---------------|---------------|-----------|
| FV    | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

Table 5-18 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

Table 5-18. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values

| VPTERNLOGD reg1, reg2, src3, 0xE2 |           |           | Bit Result with Imm8=0xE2 | VPTERNLOGD reg1, reg2, src3, 0xE4 |           |           | Bit Result with Imm8=0xE4 |
|-----------------------------------|-----------|-----------|---------------------------|-----------------------------------|-----------|-----------|---------------------------|
| Bit(reg1)                         | Bit(reg2) | Bit(src3) |                           | Bit(reg1)                         | Bit(reg2) | Bit(src3) |                           |
| 0                                 | 0         | 0         | 0                         | 0                                 | 0         | 0         | 0                         |
| 0                                 | 0         | 1         | 1                         | 0                                 | 0         | 1         | 0                         |
| 0                                 | 1         | 0         | 0                         | 0                                 | 1         | 0         | 1                         |
| 0                                 | 1         | 1         | 0                         | 0                                 | 1         | 1         | 0                         |
| 1                                 | 0         | 0         | 0                         | 1                                 | 0         | 0         | 0                         |
| 1                                 | 0         | 1         | 1                         | 1                                 | 0         | 1         | 1                         |
| 1                                 | 1         | 0         | 1                         | 1                                 | 1         | 0         | 1                         |
| 1                                 | 1         | 1         | 1                         | 1                                 | 1         | 1         | 1                         |

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

### Operation

#### VPTERNLOGD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      FOR k ← 0 TO 31

        IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

          THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]

          ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]

        FI;

      ; table lookup of immediate bellow;

    ELSE

      IF \*merging-masking\*                   ; merging-masking

        THEN \*DEST[31+i:i] remains unchanged\*

      ELSE                                   ; zeroing-masking

        DEST[31+i:i] ← 0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPTERNLOGQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      FOR k ← 0 TO 63
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
          THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
          ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
        FI;
          ; table lookup of immediate below;
      ELSE
        IF *merging-masking*
          ; merging-masking
          THEN *DEST[63+i:i] remains unchanged*
        ELSE
          ; zeroing-masking
          DEST[63+i:i] ← 0
        FI;
      FI;
    ENDFOR;
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPTERNLOGD __m512i __mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i __mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i __mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i __mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i __mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i __mm256_maskz_ternarylogic_epi32(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i __mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i __mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i __mm_maskz_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i __mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i __mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i __mm512_maskz_ternarylogic_epi64(__mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i __mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i __mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i __mm256_maskz_ternarylogic_epi64(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i __mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i __mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i __mm_maskz_ternarylogic_epi64(__mmask8 m, __m128i a, __m128i b, int imm);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F38.W0 26 /r<br>VPTESTMB k2 {k1}, xmm2,<br>xmm3/m128         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.256.66.0F38.W0 26 /r<br>VPTESTMB k2 {k1}, ymm2,<br>ymm3/m256         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.512.66.0F38.W0 26 /r<br>VPTESTMB k2 {k1}, zmm2,<br>zmm3/m512         | FVM       | V/V                          | AVX512BW                 | Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.128.66.0F38.W1 26 /r<br>VPTESTMW k2 {k1}, xmm2,<br>xmm3/m128         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.256.66.0F38.W1 26 /r<br>VPTESTMW k2 {k1}, ymm2,<br>ymm3/m256         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.512.66.0F38.W1 26 /r<br>VPTESTMW k2 {k1}, zmm2,<br>zmm3/m512         | FVM       | V/V                          | AVX512BW                 | Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.               |
| EVEX.NDS.128.66.0F38.W0 27 /r<br>VPTESTMD k2 {k1}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1. |
| EVEX.NDS.256.66.0F38.W0 27 /r<br>VPTESTMD k2 {k1}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1. |
| EVEX.NDS.512.66.0F38.W0 27 /r<br>VPTESTMD k2 {k1}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1. |
| EVEX.NDS.128.66.0F38.W1 27 /r<br>VPTESTMQ k2 {k1}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.   |
| EVEX.NDS.256.66.0F38.W1 27 /r<br>VPTESTMQ k2 {k1}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.   |
| EVEX.NDS.512.66.0F38.W1 27 /r<br>VPTESTMQ k2 {k1}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.   |

## Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FVM   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

## Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

**VPTESTMD/VPTESTMQ:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

**VPTESTMB/VPTESTMW:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

## Operation

**VPTESTMB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[j] ← (SRC1[j+7:i] BITWISE AND SRC2[j+7:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPTESTMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[j] ← (SRC1[j+15:i] BITWISE AND SRC2[j+15:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPTESTMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] ← (SRC1[j+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
        ELSE DEST[j] ← (SRC1[j+31:i] BITWISE AND SRC2[j+31:i] != 0)? 1 : 0;
      FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
```

DEST[MAX\_KL-1:KL] ← 0

#### **VPTESTMQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;

        ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;

      FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

#### **Intel C/C++ Compiler Intrinsic Equivalents**

VPTESTMB \_\_mmask64 \_mm512\_test\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMB \_\_mmask64 \_mm512\_mask\_test\_epi8\_mask(\_\_mmask64, \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32 \_mm512\_test\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32 \_mm512\_mask\_test\_epi16\_mask(\_\_mmask32, \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16 \_mm512\_test\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16 \_mm512\_mask\_test\_epi32\_mask(\_\_mmask16, \_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8 \_mm512\_test\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8 \_mm512\_mask\_test\_epi64\_mask(\_\_mmask8, \_\_m512i a, \_\_m512i b);

#### **SIMD Floating-Point Exceptions**

None

#### **Other Exceptions**

VPTESTMD/Q: See Exceptions Type E4.

VPTESTMB/W: See Exceptions Type E4.nb.

## VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F38.W0 46 /r<br>VPSRAVD xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX2                     | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.                            |
| VEX.NDS.256.66.0F38.W0 46 /r<br>VPSRAVD ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.                            |
| EVEX.NDS.128.66.0F38.W1 11 /r<br>VPSRAVW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.               |
| EVEX.NDS.256.66.0F38.W1 11 /r<br>VPSRAVW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | FVM       | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.               |
| EVEX.NDS.512.66.0F38.W1 11 /r<br>VPSRAVW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | FVM       | V/V                          | AVX512BW                 | Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.               |
| EVEX.NDS.128.66.0F38.W0 46 /r<br>VPSRAVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1. |
| EVEX.NDS.256.66.0F38.W0 46 /r<br>VPSRAVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1. |
| EVEX.NDS.512.66.0F38.W0 46 /r<br>VPSRAVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1. |
| EVEX.NDS.128.66.0F38.W1 46 /r<br>VPSRAVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.   |
| EVEX.NDS.256.66.0F38.W1 46 /r<br>VPSRAVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.   |
| EVEX.NDS.512.66.0F38.W1 46 /r<br>VPSRAVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.   |

### 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        |
| FVM   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

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 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element are filled with the corresponding 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 16 (for word), 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 (MAX\_VL-1:128) of the corresponding destination 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. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation****VPSRAVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      COUNT ← SRC2[i+3:i]
      IF COUNT < 16
        THEN DEST[i+15:i] ← SignExtend(SRC1[i+15:i] >> COUNT)
        ELSE
          FOR k ← 0 TO 15
            DEST[i+k] ← SRC1[i+15]
          ENDFOR;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+15:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
          FI
        FI;
      ENDFOR;
    DEST[MAX_VL-1:VL] ← 0;

```



**VPSRAVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAX_VL-1:128] ← 0;

```

**VPSRAVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAX_VL-1:256] ← 0;

```

**VPSRAVD (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        COUNT ← SRC2[4:0]
        IF COUNT < 32
          THEN DEST[j+31:i] ← SignExtend(SRC1[j+31:i] >> COUNT)
          ELSE
            FOR k ← 0 TO 31
              DEST[j+k] ← SRC1[j+31]
            ENDFOR;
          FI
        ELSE
          COUNT ← SRC2[i+4:i]
          IF COUNT < 32
            THEN DEST[j+31:i] ← SignExtend(SRC1[j+31:i] >> COUNT)
            ELSE
              FOR k ← 0 TO 31
                DEST[j+k] ← SRC1[j+31]
              ENDFOR;
            FI
          FI;
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
              DEST[31:0] ← 0
            FI
          FI;
        ENDFOR;
      DEST[MAX_VL-1:VL] ← 0;

```

**VPSRAVQ (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

COUNT ← SRC2[5:0]

IF COUNT &lt; 64

THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k ← 0 TO 63

DEST[i+k] ← SRC1[i+63]

ENDFOR;

FI

ELSE

COUNT ← SRC2[i+5:i]

IF COUNT &lt; 64

THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k ← 0 TO 63

DEST[i+k] ← SRC1[i+63]

ENDFOR;

FI

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRAVD \_\_m512i \_\_mm512\_srav\_epi32(\_\_m512i a, \_\_m512i cnt);

VPSRAVD \_\_m512i \_\_mm512\_mask\_srav\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVD \_\_m512i \_\_mm512\_maskz\_srav\_epi32(\_\_mmask16 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVD \_\_m256i \_\_mm256\_srav\_epi32(\_\_m256i a, \_\_m256i cnt);

VPSRAVD \_\_m256i \_\_mm256\_mask\_srav\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVD \_\_m256i \_\_mm256\_maskz\_srav\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVD \_\_m128i \_\_mm\_srav\_epi32(\_\_m128i a, \_\_m128i cnt);

VPSRAVD \_\_m128i \_\_mm\_mask\_srav\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVD \_\_m128i \_\_mm\_maskz\_srav\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVQ \_\_m512i \_\_mm512\_srav\_epi64(\_\_m512i a, \_\_m512i cnt);

VPSRAVQ \_\_m512i \_\_mm512\_mask\_srav\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVQ \_\_m512i \_\_mm512\_maskz\_srav\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVQ \_\_m256i \_\_mm256\_srav\_epi64(\_\_m256i a, \_\_m256i cnt);

VPSRAVQ \_\_m256i \_\_mm256\_mask\_srav\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVQ \_\_m256i \_\_mm256\_maskz\_srav\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVQ \_\_m128i \_\_mm\_srav\_epi64(\_\_m128i a, \_\_m128i cnt);

VPSRAVQ \_\_m128i \_\_mm\_mask\_srav\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVQ \_\_m128i \_\_mm\_maskz\_srav\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVW \_\_m512i \_\_mm512\_srav\_epi16(\_\_m512i a, \_\_m512i cnt);  
 VPSRAVW \_\_m512i \_\_mm512\_mask\_srav\_epi16(\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i cnt);  
 VPSRAVW \_\_m512i \_\_mm512\_maskz\_srav\_epi16(\_\_mmask32 m, \_\_m512i a, \_\_m512i cnt);  
 VPSRAVW \_\_m256i \_\_mm256\_srav\_epi16(\_\_m256i a, \_\_m256i cnt);  
 VPSRAVW \_\_m256i \_\_mm256\_mask\_srav\_epi16(\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i cnt);  
 VPSRAVW \_\_m256i \_\_mm256\_maskz\_srav\_epi16(\_\_mmask16 m, \_\_m256i a, \_\_m256i cnt);  
 VPSRAVW \_\_m128i \_\_mm\_srav\_epi16(\_\_m128i a, \_\_m128i cnt);  
 VPSRAVW \_\_m128i \_\_mm\_mask\_srav\_epi16(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);  
 VPSRAVW \_\_m128i \_\_mm\_maskz\_srav\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);  
 VPSRAVD \_\_m256i \_\_mm256\_srav\_epi32 (\_\_m256i m, \_\_m256i count)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**PXOR/PXORD/PXORQ—Exclusive Or**

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| 66 0F EF /r<br>PXOR xmm1, xmm2/m128  | RM        | V/V                          | SSE2                     | Bitwise XOR of xmm2/m128 and xmm1.   |
| VEX.NDS.128.66.0F.WIG EF /r<br>VPXOR xmm1, xmm2, xmm3/m128                     | RVM       | V/V                          | AVX                      | Bitwise XOR of xmm3/m128 and xmm2.   |
| VEX.NDS.256.66.0F.WIG EF /r<br>VPXOR ymm1, ymm2, ymm3/m256                     | RVM       | V/V                          | AVX2                     | Bitwise XOR of ymm3/m256 and ymm2.   |
| EVEX.NDS.128.66.0F.W0 EF /r<br>VPXORD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise XOR of packed doubleword integers in xmm2<br>and xmm3/m128 using writemask k1.         |
| EVEX.NDS.256.66.0F.W0 EF /r<br>VPXORD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise XOR of packed doubleword integers in ymm2<br>and ymm3/m256 using writemask k1.         |
| EVEX.NDS.512.66.0F.W0 EF /r<br>VPXORD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV        | V/V                          | AVX512F                  | Bitwise XOR of packed doubleword integers in zmm2<br>and zmm3/m512/m32bcst using writemask k1. |
| EVEX.NDS.128.66.0F.W1 EF /r<br>VPXORQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise XOR of packed quadword integers in xmm2<br>and xmm3/m128 using writemask k1.           |
| EVEX.NDS.256.66.0F.W1 EF /r<br>VPXORQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Bitwise XOR of packed quadword integers in ymm2<br>and ymm3/m256 using writemask k1.           |
| EVEX.NDS.512.66.0F.W1 EF /r<br>VPXORQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV        | V/V                          | AVX512F                  | Bitwise XOR of packed quadword integers in zmm2<br>and zmm3/m512/m64bcst using writemask k1.   |

**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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Performs a bitwise logical XOR operation on the second source operand and the first source operand and stores the result in the destination operand. Each bit of the result is set to 0 if the corresponding bits of the first and second operands are identical; otherwise, it is set to 0.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

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: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

## Operation

### PXOR (Legacy SSE instruction)

DEST[127:0] ← (DEST[127:0] BITWISE XOR SRC[127:0])

### VPXOR (VEX.128 encoded instruction)

DEST[127:0] ← (SRC1[127:0] BITWISE XOR SRC2[127:0])

DEST[MAX\_VL-1:128] ← 0

### VPXOR (VEX.256 encoded instruction)

DEST[255:0] ← (SRC1[255:0] BITWISE XOR SRC2[255:0])

DEST[MAX\_VL-1:256] ← 0

### VPXORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0]

      ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[31:0] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[31:0] ← 0

  FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPXORQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPXORD \_\_m512i \_\_mm512\_xor\_epi32(\_\_m512i a, \_\_m512i b)

VPXORD \_\_m512i \_\_mm512\_mask\_xor\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a, \_\_m512i b)

VPXORD \_\_m512i \_\_mm512\_maskz\_xor\_epi32(\_\_mmask16 m, \_\_m512i a, \_\_m512i b)

VPXORD \_\_m256i \_\_mm256\_xor\_epi32(\_\_m256i a, \_\_m256i b)

VPXORD \_\_m256i \_\_mm256\_mask\_xor\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b)

VPXORD \_\_m256i \_\_mm256\_maskz\_xor\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i b)

VPXORD \_\_m128i \_\_mm\_xor\_epi32(\_\_m128i a, \_\_m128i b)

VPXORD \_\_m128i \_\_mm\_mask\_xor\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPXORD \_\_m128i \_\_mm\_maskz\_xor\_epi32(\_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPXORQ \_\_m512i \_\_mm512\_xor\_epi64(\_\_m512i a, \_\_m512i b);

VPXORQ \_\_m512i \_\_mm512\_mask\_xor\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i b);

VPXORQ \_\_m512i \_\_mm512\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i b);

VPXORQ \_\_m256i \_\_mm256\_xor\_epi64(\_\_m256i a, \_\_m256i b);

VPXORQ \_\_m256i \_\_mm256\_mask\_xor\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPXORQ \_\_m256i \_\_mm256\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPXORQ \_\_m128i \_\_mm\_xor\_epi64(\_\_m128i a, \_\_m128i b);

VPXORQ \_\_m128i \_\_mm\_mask\_xor\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b);

VPXORQ \_\_m128i \_\_mm\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

PXOR \_\_m128i \_\_mm\_xor\_si128 (\_\_m128i a, \_\_m128i b)

VPXOR \_\_m256i \_\_mm256\_xor\_si256 (\_\_m256i a, \_\_m256i b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F3A.W1 50 /r ib<br>VRANGEPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.   |
| EVEX.NDS.256.66.0F3A.W1 50 /r ib<br>VRANGEPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Calculate four RANGE operation output value from 4pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.   |
| EVEX.NDS.512.66.0F3A.W1 50 /r ib<br>VRANGEPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{sae}, imm8 | FV        | V/V                          | AVX512DQ                 | Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-40.

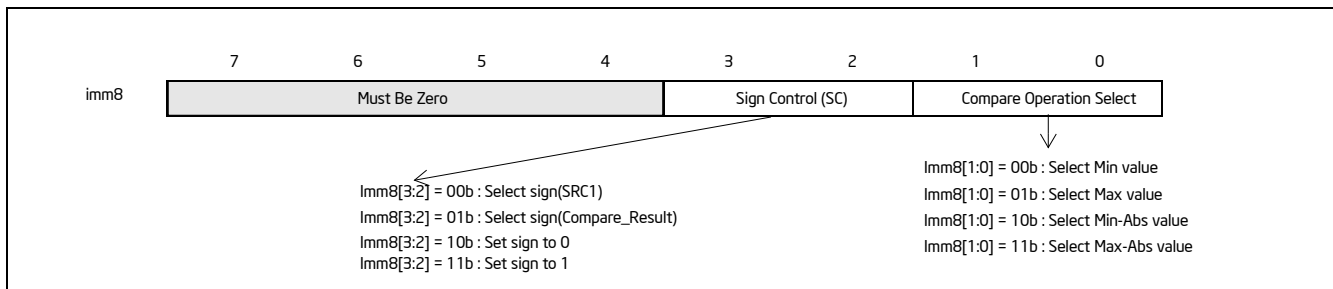


Figure 5-40. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

**Table 5-19. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]**

| Src1  | Src2  | Result       | IE Signaling Due to Comparison | Imm8[3:2] Effect to Range Output |
|-------|-------|--------------|--------------------------------|----------------------------------|
| sNaN1 | sNaN2 | Quiet(sNaN1) | Yes                            | Ignored                          |
| sNaN1 | qNaN2 | Quiet(sNaN1) | Yes                            | Ignored                          |
| sNaN1 | Norm2 | Quiet(sNaN1) | Yes                            | Ignored                          |
| qNaN1 | sNaN2 | Quiet(sNaN2) | Yes                            | Ignored                          |
| qNaN1 | qNaN2 | qNaN1        | No                             | Applicable                       |
| qNaN1 | Norm2 | Norm2        | No                             | Applicable                       |
| Norm1 | sNaN2 | Quiet(sNaN2) | Yes                            | Ignored                          |
| Norm1 | qNaN2 | Norm1        | No                             | Applicable                       |

**Table 5-20. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN\_ABS and MAX, MAX\_ABS**

| MIN and MIN_ABS |      |        | MAX and MAX_ABS |      |        |
|-----------------|------|--------|-----------------|------|--------|
| Src1            | Src2 | Result | Src1            | Src2 | Result |
| +0              | -0   | -0     | +0              | -0   | +0     |
| -0              | +0   | -0     | -0              | +0   | +0     |

**Table 5-21. Comparison Result of Equal-Magnitude Input Cases for MIN\_ABS and MAX\_ABS, (|a| = |b|, a > 0, b < 0)**

| MIN_ABS ( a  =  b , a > 0, b < 0) |      |        | MAX_ABS ( a  =  b , a > 0, b < 0) |      |        |
|-----------------------------------|------|--------|-----------------------------------|------|--------|
| Src1                              | Src2 | Result | Src1                              | Src2 | Result |
| a                                 | b    | b      | a                                 | b    | a      |
| b                                 | a    | b      | b                                 | a    | a      |

**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCt[1:0], SignSelCt[1:0])

```
{
    // Check if SNAN and report IE, see also Table 5-19
    IF (SRC1 = SNAN) THEN RETURN (QNaN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNaN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN // Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNaN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
```



```

IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN// Src2 is a denormal number
  IF DAZ THEN Src2.fraction ← 0;
  ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-20
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-21
ELSE
  Case(CmpOpCtl[1:0])
  00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
  01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
  10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
  11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
  ESAC;
FI;

Case(SignSelCtl[1:0])
00: dest ← (SRC1[63] << 63) OR (TMP[62:0]); // Preserve Src1 sign bit
01: dest ← TMP[63:0]; // Preserve sign of compare result
10: dest ← (0 << 63) OR (TMP[62:0]); // Zero out sign bit
11: dest ← (1 << 63) OR (TMP[62:0]); // Set the sign bit
ESAC;
RETURN dest[63:0];
}

```

```

CmpOpCtl[1:0] = imm8[1:0];
SignSelCtl[1:0] = imm8[3:2];

```

### VRANGEPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

      ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], DAZ, CmpOpCtl[1:0], SignSelCtl[1:0]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] = 0

  FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;
```

Where:

`zmm_dst` is the destination operand.

`zmm_src` is the input operand to compare against  $\pm 1023$  (this is SRC1).

`zmm_1023` is the reference operand, contains the value of 1023 (and this is SRC2).

`IMM=02(imm8[1:0]='10)` selects the Min Absolute value operation with selection of SRC1.sign.

In case  $|zmm\_src| < 1023$  (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into `zmm_dst`. Otherwise, the value stored in `zmm_dst` will get the value of 1023 (received on `zmm_1023`, which is SRC2).

However, the sign control (`imm8[3:2]='00`) instructs to select the sign of SRC1 received from `zmm_src`. So, even in the case of  $|zmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if `zmm_src < -1023`, the result of `VRANGEPD` will be the minimal value of -1023 while if `zmm_src > +1023`, the result of `VRANGE` will be the maximal value of +1023.

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGEPD __m512d __mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d __mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d __mm512_mask_range_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d __mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d __mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d __mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d __mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d __mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d __mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d __mm_range_pd (__m128d a, __m128d b, int imm);
VRANGEPD __m128d __mm_mask_range_pd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d __mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E2.

## VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F3A.W0 50 /r ib<br>VRANGEPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.  |
| EVEX.NDS.256.66.0F3A.W0 50 /r ib<br>VRANGEPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation. |
| EVEX.NDS.512.66.0F3A.W0 50 /r ib<br>VRANGEPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{sae}, imm8 | FV        | V/V                          | AVX512DQ                 | Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-40.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

### Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
```

```
{
    // Check if SNAN and report IE, see also Table 5-19
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);
```

```

Src1.exp ← SRC1[30:23];
Src1.fraction ← SRC1[22:0];
IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
FI;
Src2.exp ← SRC2[30:23];
Src2.fraction ← SRC2[22:0];
IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-20
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-21
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
    01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
    10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
    11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
    ESAC;
FI;
Case(SignSelCtl[1:0])
00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
01: dest ← TMP[31:0];// Preserve sign of compare result
10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
ESAC;
RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGEPS**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

ELSE DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[i+31:i], DAZ, CmpOpCtl[1:0], SignSelCtl[1:0]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

```
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|zmm\_src| < 150$ , then its value will be written into zmm\_dst. Otherwise, the value stored in zmm\_dst will get the value of 150 (received on zmm\_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm\_src. So, even in the case of  $|zmm\_src| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if zmm\_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm\_src > +150, the result of VRANGE will be the maximal value of +150.

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGEPS __m512 __mm512_range_ps ( __m512 a, __m512 b, int imm);
VRANGEPS __m512 __mm512_range_round_ps ( __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 __mm512_mask_range_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 __mm512_mask_range_round_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 __mm512_maskz_range_ps ( __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 __mm512_maskz_range_round_ps ( __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 __mm256_range_ps ( __m256 a, __m256 b, int imm);
VRANGEPS __m256 __mm256_mask_range_ps ( __m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 __mm256_maskz_range_ps ( __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 __mm_range_ps ( __m128 a, __m128 b, int imm);
VRANGEPS __m128 __mm_mask_range_ps ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 __mm_maskz_range_ps ( __mmask8 k, __m128 a, __m128 b, int imm);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E2.

## VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F3A.W1 51 /r<br>VRANGESD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae}, imm8 | T1S       | V/V                          | AVX512DQ                 | Calculate a RANGE operation output value from 2 double-precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

### Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-40.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2] has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

### Operation

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
  // Check if SNAN and report IE, see also Table 5-19
  IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[62:52];
  Src1.fraction ← SRC1[51:0];
  IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;
```

```

Src2.exp ← SRC2[62:52];
Src2.fraction ← SRC2[51:0];
IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-20
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-21
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
    01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
    10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
    11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
    ESAC;
FI;

Case(SignSelCtl[1:0])
00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
01: dest ← TMP[63:0];// Preserve sign of compare result
10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
ESAC;
RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGESD**

```

IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking* ;merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ;zeroing-masking
                DEST[63:0] = 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;
```

Where:

xmm\_dst is the destination operand.

xmm\_src is the input operand to compare against  $\pm 1023$ .

xmm\_1023 is the reference operand, contains the value of 1023.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|xmm\_src| < 1023$ , then its value will be written into xmm\_dst. Otherwise, the value stored in xmm\_dst will get the value of 1023 (received on xmm\_1023).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm\_src. So, even in the case of  $|xmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if  $xmm\_src < -1023$ , the result of VRANGEPD will be the minimal value of -1023 while if  $xmm\_src > +1023$ , the result of VRANGE will be the maximal value of +1023.

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGESD __m128d _mm_range_sd (__m128d a, __m128d b, int imm);
```

```
VRANGESD __m128d _mm_range_round_sd (__m128d a, __m128d b, int imm, int sae);
```

```
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
```

```
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

```
VRANGESD __m128d _mm_maskz_range_sd (__mmask8 k, __m128d a, __m128d b, int imm);
```

```
VRANGESD __m128d _mm_maskz_range_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

#### SIMD Floating-Point Exceptions

Invalid, Denormal

#### Other Exceptions

See Exceptions Type E3.



## VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F3A.W0 51 /r<br>VRANGESS xmm1 {k1}{z},<br>xmm2, xmm3/m32{sae}, imm8 | T1S       | V/V                          | AVX512DQ                 | Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-40.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

### Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
// Check if SNAN and report IE, see also Table 5-19
IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

Src1.exp ← SRC1[30:23];
Src1.fraction ← SRC1[22:0];
IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
FI;
Src2.exp ← SRC2[30:23];
```

```

Src2.fraction ← SRC2[22:0];
IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-20
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-21
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
    01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
    10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
    11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
    ESAC;
FI;
Case(SignSelCtl[1:0])
00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
01: dest ← TMP[31:0];// Preserve sign of compare result
10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
ESAC;
RETURN dest[31:0];
}

```

```

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGESS**

```

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        DEST[31:0] = 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGESS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|\text{xmm\_src}| < 150$ , then its value will be written into `zmm_dst`. Otherwise, the value stored in `xmm_dst` will get the value of 150 (received on `zmm_150`).

However, the sign control (`imm8[3:2]='00`) instructs to select the sign of SRC1 received from `xmm_src`. So, even in the case of  $|\text{xmm\_src}| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if `xmm_src`  $< -150$ , the result of `VRANGESS` will be the minimal value of -150 while if `xmm_src`  $> +150$ , the result of `VRANGE` will be the maximal value of +150.

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGESS __m128 _mm_range_ss ( __m128 a, __m128 b, int imm);
```

```
VRANGESS __m128 _mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
```

```
VRANGESS __m128 _mm_mask_range_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
```

```
VRANGESS __m128 _mm_mask_range_round_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
```

```
VRANGESS __m128 _mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
```

```
VRANGESS __m128 _mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E3.

## VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 4C /r<br>VRCP14PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocals of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask. |
| EVEX.256.66.0F38.W1 4C /r<br>VRCP14PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocals of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask. |
| EVEX.512.66.0F38.W1 4C /r<br>VRCP14PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst | FV        | V/V                          | AVX512F                  | Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

#### Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-22. VRCP14PD/VRCP14SD Special Cases**

| Input value                 | Result value | Comments                                 |
|-----------------------------|--------------|--|
| $0 \leq X \leq 2^{-1024}$   | INF          | Very small denormal                      |
| $-2^{-1024} \leq X \leq -0$ | -INF         | Very small denormal                      |
| $X > 2^{1022}$              | Underflow    | Up to 18 bits of fractions are returned* |
| $X < -2^{1022}$             | -Underflow   | Up to 18 bits of fractions are returned* |
| $X = 2^{-n}$                | $2^n$        |  |
| $X = -2^{-n}$               | $-2^n$       |  |

\* in this case the mantissa is shifted right by one or two bits

**Operation****VRCP14PD ((EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← APPROXIMATE(1.0/SRC[63:0]);
      ELSE DEST[i+63:i] ← APPROXIMATE(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PD \_\_m512d \_\_mm512\_rcp14\_pd( \_\_m512d a);

VRCP14PD \_\_m512d \_\_mm512\_mask\_rcp14\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VRCP14PD \_\_m512d \_\_mm512\_maskz\_rcp14\_pd( \_\_mmask8 k, \_\_m512d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

| Opcode/<br>Instruction  | Op<br>/ En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F38.W1 4D /r<br>VRCP14SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64 | T1S        | V/V                          | AVX512F                  | Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-22 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

### Operation

#### VRCP14SD (EVEX version)

```

IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[63:0] ← 0
FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VRCP14SD __m128d _mm_rcp14_sd( __m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd( __mmask8 k, __m128d a, __m128d b);

```

#### SIMD Floating-Point Exceptions

None

**Other Exceptions**

See Exceptions Type E5.

## VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 4C /r<br>VRCP14PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask. |
| EVEX.256.66.0F38.W0 4C /r<br>VRCP14PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask. |
| EVEX.512.66.0F38.W0 4C /r<br>VRCP14PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst | FV        | V/V                          | AVX512F                  | Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-23. VRCP14PS/VRCP14SS Special Cases**

| Input value                | Result value | Comments                                 |
|----------------------------|--------------|--|
| $0 \leq X \leq 2^{-128}$   | INF          | Very small denormal                      |
| $-2^{-128} \leq X \leq -0$ | -INF         | Very small denormal                      |
| $X > 2^{126}$              | Underflow    | Up to 18 bits of fractions are returned* |
| $X < -2^{126}$             | -Underflow   | Up to 18 bits of fractions are returned* |
| $X = 2^{-n}$               | $2^n$        |  |
| $X = -2^{-n}$              | $-2^n$       |  |

\* in this case the mantissa is shifted right by one or two bits



**Operation****VRCP14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/SRC[31:0]);

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/SRC[i+31:i]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PS \_\_m512 \_\_mm512\_rcp14\_ps( \_\_m512 a);

VRCP14PS \_\_m512 \_\_mm512\_mask\_rcp14\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VRCP14PS \_\_m512 \_\_mm512\_maskz\_rcp14\_ps( \_\_mmask16 k, \_\_m512 a);

VRCP14PS \_\_m256 \_\_mm256\_rcp14\_ps( \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_mask\_rcp14\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_maskz\_rcp14\_ps( \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m128 \_\_mm\_rcp14\_ps( \_\_m128 a);

VRCP14PS \_\_m128 \_\_mm\_mask\_rcp14\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VRCP14PS \_\_m128 \_\_mm\_maskz\_rcp14\_ps( \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F38.W0 4D /r<br>VRCP14SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32 | T1S       | V/V                          | AVX512F                  | Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-23 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

### Operation

#### VRCP14SS (EVEX version)

```

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[31:0] ← 0
FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VRCP14SS __m128 __mm_rcp14_ss( __m128 a, __m128 b);
VRCP14SS __m128 __mm_mask_rcp14_ss( __m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 __mm_maskz_rcp14_ss( __mmask8 k, __m128 a, __m128 b);

```

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

See Exceptions Type E5.

## VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 56 /r ib<br>VREDUCEPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1. |
| EVEX.256.66.0F3A.W1 56 /r ib<br>VREDUCEPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1. |
| EVEX.512.66.0F3A.W1 56 /r ib<br>VREDUCEPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae},<br>imm8 | FV        | V/V                          | AVX512DQ                 | Perform reduction transformation on double-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.        |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-41. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}2$ ,

where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

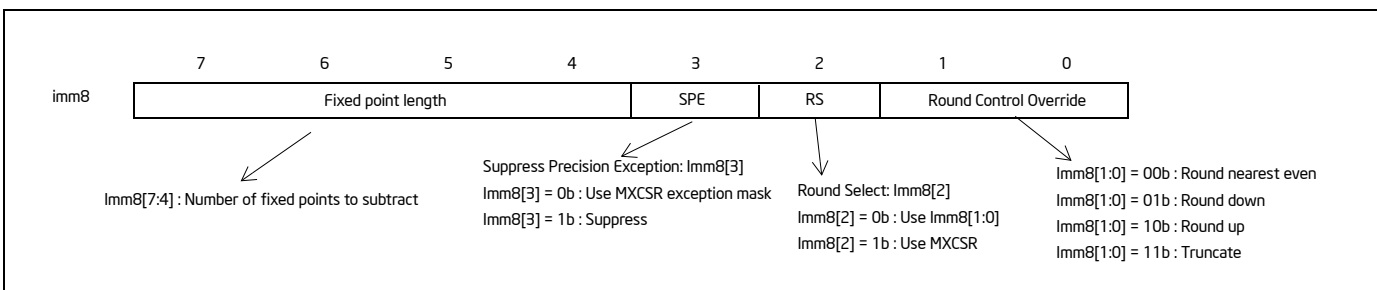


Figure 5-41. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-24.

**Table 5-24. VREDUCEPD/SD/PS/SS Special Cases**

|  | Round Mode    | Returned value                  |
|--|---------------|---------------------------------|
| $ \text{Src1}  < 2^{-M-1}$             | RNE           | Src1                            |
| $ \text{Src1}  < 2^{-M}$               | RPI, Src1 > 0 | Round (Src1-2 <sup>-M</sup> ) * |
|  | RPI, Src1 ≤ 0 | Src1                            |
|  | RNI, Src1 ≥ 0 | Src1                            |
|  | RNI, Src1 < 0 | Round (Src1+2 <sup>-M</sup> ) * |
| Src1 = ±0, or<br>Dest = ±0 (Src1!=INF) | NOT RNI       | +0.0                            |
|  | RNI           | -0.0                            |
| Src1 = ±INF                            | any           | +0.0                            |
| Src1 = ±NAN                            | n/a           | QNaN(Src1)                      |

\* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC_source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[63:0] ← 2-M * {ROUND(2M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
  TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

### VREDUCEPD

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b == 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] ← ReduceArgumentDP(SRC[63:0], imm8[7:0]);

      ELSE DEST[i+63:i] ← ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] = 0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPD \_\_m512d \_\_mm512\_mask\_reduce\_pd( \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m512d \_\_mm512\_mask\_reduce\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m512d \_\_mm512\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m256d \_\_mm256\_mask\_reduce\_pd( \_\_m256d a, int imm)  
 VREDUCEPD \_\_m256d \_\_mm256\_mask\_reduce\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int imm)  
 VREDUCEPD \_\_m256d \_\_mm256\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m256d a, int imm)  
 VREDUCEPD \_\_m128d \_\_mm\_mask\_reduce\_pd( \_\_m128d a, int imm)  
 VREDUCEPD \_\_m128d \_\_mm\_mask\_reduce\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int imm)  
 VREDUCEPD \_\_m128d \_\_mm\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m128d a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD                      If EVEX.vvvv != 1111B.

## VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32 bit<br>Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F3A.W1 57<br>/r<br>VREDUCESD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae},<br>imm8 | T1S        | V/V                          | AVX512D<br>Q             | Perform a reduction transformation on a scalar double-precision floating point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-41. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ ,

where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-24.

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC_source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[63:0] ← 2-M * {ROUND(2M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
  TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESD**

```

IF k1[0] or *no writemask*
  THEN  DEST[63:0] ← ReduceArgumentDP(SRC2[63:0], imm8[7:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[63:0] = 0
    FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCESD __m128d _mm_mask_reduce_sd( __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.



## VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 56 /r ib<br>VREDUCEPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Perform reduction transformation on packed single-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1. |
| EVEX.256.66.0F3A.W0 56 /r ib<br>VREDUCEPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8         | FV        | V/V                          | AVX512VL<br>AVX512DQ     | Perform reduction transformation on packed single-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1. |
| EVEX.512.66.0F3A.W0 56 /r ib<br>VREDUCEPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{saе},<br>imm8 | FV        | V/V                          | AVX512DQ                 | Perform reduction transformation on packed single-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-41. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-24.

### Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
```

```

SPE ← 0; // Suppress Precision Exception
TMP[31:0] ← 2-M * {ROUND(2M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}

```

**VREDUCEPS**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← ReduceArgumentSP(SRC[31:0], imm8[7:0]);

ELSE DEST[i+31:i] ← ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPS \_\_m512 \_\_mm512\_mask\_reduce\_ps(\_\_m512 a, int imm, int sae)

VREDUCEPS \_\_m512 \_\_mm512\_mask\_reduce\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, int imm, int sae)

VREDUCEPS \_\_m512 \_\_mm512\_maskz\_reduce\_ps(\_\_mmask16 k, \_\_m512 a, int imm, int sae)

VREDUCEPS \_\_m256 \_\_mm256\_mask\_reduce\_ps(\_\_m256 a, int imm)

VREDUCEPS \_\_m256 \_\_mm256\_mask\_reduce\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, int imm)

VREDUCEPS \_\_m256 \_\_mm256\_maskz\_reduce\_ps(\_\_mmask8 k, \_\_m256 a, int imm)

VREDUCEPS \_\_m128 \_\_mm\_mask\_reduce\_ps(\_\_m128 a, int imm)

VREDUCEPS \_\_m128 \_\_mm\_mask\_reduce\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, int imm)

VREDUCEPS \_\_m128 \_\_mm\_maskz\_reduce\_ps(\_\_mmask8 k, \_\_m128 a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD If EVEX.vvvv != 1111B.

## VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F3A.W0 57<br>/r /ib<br>VREDUCESS xmm1 {k1}{z},<br>xmm2, xmm3/m32{sae},<br>imm8 | T1S       | V/V                         | AVX512DQ                 | Perform a reduction transformation on a scalar single-precision floating point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-41. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ ,

where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-24.

### Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[31:0] ← 2-M * {ROUND(2M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
  TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESS**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← ReduceArgumentSP(SRC2[31:0], imm8[7:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] = 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCESS __m128 __mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 __mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 __mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W1 09 /r ib<br>VRNDSCALEPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512F      | Rounds packed double-precision floating point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.   |
| EVEX.256.66.0F3A.W1 09 /r ib<br>VRNDSCALEPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512F      | Rounds packed double-precision floating point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.   |
| EVEX.512.66.0F3A.W1 09 /r ib<br>VRNDSCALEPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae}, imm8 | FV        | V/V                          | AVX512F                  | Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-42) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the 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 (Immediate control table below 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.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

$$\begin{aligned} \text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \end{aligned}$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

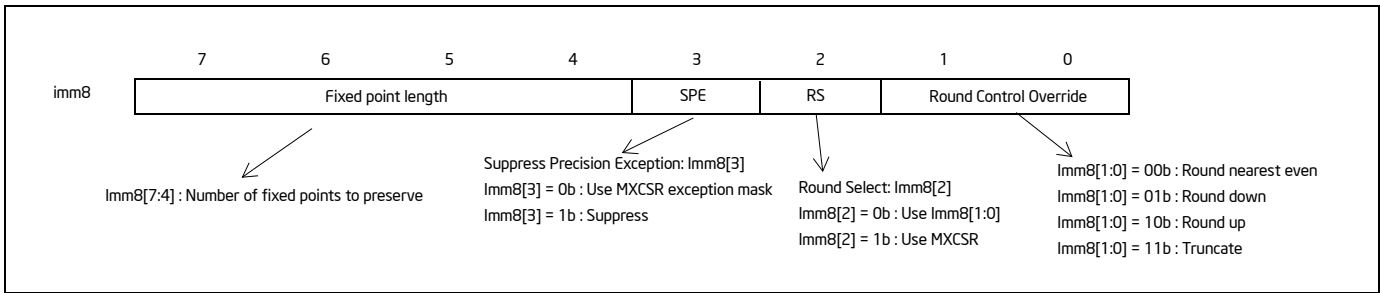


Figure 5-42. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-25.

Table 5-25. VRNDSCALEPD/SD/PS/SS Special Cases

|           | Returned value         |
|-----------|------------------------|
| Src1=±inf | Src1                   |
| Src1=±NAN | Src1 converted to QNAN |
| Src1=±0   | Src1                   |

**Operation**

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2-M* TMP[63:0] ; scale down back to 2-M

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}
    
```

**VRNDSCALEPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF \*src is a memory operand\*

THEN TMP\_SRC ← BROADCAST64(SRC, VL, k1)

ELSE TMP\_SRC ← SRC

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← RoundToIntegerDP((TMP\_SRC[i+63:i], imm8[7:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALEPD \_\_m512d \_\_mm512\_roundscale\_pd( \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_roundscale\_round\_pd( \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m512d \_\_mm512\_mask\_roundscale\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_mask\_roundscale\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m512d \_\_mm512\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_maskz\_roundscale\_round\_pd( \_\_mmask8 k, \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m256d \_\_mm256\_roundscale\_pd( \_\_m256d a, int imm);

VRNDSCALEPD \_\_m256d \_\_mm256\_mask\_roundscale\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int imm);

VRNDSCALEPD \_\_m256d \_\_mm256\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m256d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_roundscale\_pd( \_\_m128d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_mask\_roundscale\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m128d a, int imm);

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.

**VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits**

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F3A.W1 0B /r ib<br>VRNDSCALESD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{sae}, imm8 | T1S       | V/V                          | AVX512F                  | Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | Imm8      |

**Description**

Rounds a double-precision floating-point value in the low quadword (see Figure 5-42) element the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the third operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAX\_VL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the 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 (Immediate control table below 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.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

$$\begin{aligned} \text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \end{aligned}$$

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-25.



**Operation**

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR.RC      ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]     ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]                          ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2-M* TMP[63:0]           ; scale down back to 2-M

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

```

VRNDSCALESD (EVEX encoded version)

```

IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESD __m128d __mm_roundscale_sd (__m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_roundscale_round_sd (__m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d __mm_mask_roundscale_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_mask_roundscale_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d __mm_maskz_roundscale_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_maskz_roundscale_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 08 /r ib<br>VRNDSCALEPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512F      | Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask. |
| EVEX.256.66.0F3A.W0 08 /r ib<br>VRNDSCALEPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8      | FV        | V/V                          | AVX512VL<br>AVX512F      | Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask. |
| EVEX.512.66.0F3A.W0 08 /r ib<br>VRNDSCALEPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae}, imm8 | FV        | V/V                          | AVX512F                  | Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.  |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | Imm8      | NA        |

### Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-42) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the 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 (Immediate control table below 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.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.  
Handling of special case of input values are listed in Table 5-25.

### Operation

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction ← MXCSR.RC      ; get round control from MXCSR
  else
    rounding_direction ← imm8[1:0]     ; get round control from imm8[1:0]
  FI
  M ← imm8[7:4]                       ; get the scaling factor

  case (rounding_direction)
  00: TMP[31:0] ← round_to_nearest_even_integer(2M*SRC[31:0])
  01: TMP[31:0] ← round_to_equal_or_smaller_integer(2M*SRC[31:0])
  10: TMP[31:0] ← round_to_equal_or_larger_integer(2M*SRC[31:0])
  11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
  ESAC;

  Dest[31:0] ← 2-M* TMP[31:0]         ; scale down back to 2-M
  if (imm8[3] = 0) Then                ; check SPE
    if (SRC[31:0] != Dest[31:0]) Then  ; check precision lost
      set_precision()                 ; set #PE
    FI;
  FI;
  return(Dest[31:0])
}

```

VRNDSCALEPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF \*src is a memory operand\*

THEN TMP\_SRC ← BROADCAST32(SRC, VL, k1)

ELSE TMP\_SRC ← SRC

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← RoundToIntegerSP(TMP\_SRC[i+31:i], imm8[7:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALEPS __m512 __mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 __mm256_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_maskz_roundscale_ps( __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 __mm_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m128 __mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 __mm_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.

## VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F3A.W0 0A /r ib<br>VRNDSCALESS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae}, imm8 | T1S       | V/V                          | AVX512F                  | Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-42) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAX\_VL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the 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 (Immediate control tables below 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.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-25.

**Operation**

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
    ESAC;

    Dest[31:0] ← 2-M* TMP[31:0] ; scale down back to 2-M
    if (imm8[3] = 0) Then ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

```

VRNDSCALESS (EVEX encoded version)

```

IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESS __m128 _mm_roundscale_ss (__m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss (__m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 4E /r<br>VRSQRT14PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocal square roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask. |
| EVEX.256.66.0F38.W1 4E /r<br>VRSQRT14PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocal square roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask. |
| EVEX.512.66.0F38.W1 4E /r<br>VRSQRT14PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst | FV        | V/V                          | AVX512F                  | Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.  |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ .

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_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.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****VRSQRT14PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[63:0]));

ELSE DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Table 5-26. VRSQRT14PD Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +0              |                          |

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PD \_\_m512d \_\_mm512\_rsqrt14\_pd( \_\_m512d a);

VRSQRT14PD \_\_m512d \_\_mm512\_mask\_rsqrt14\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VRSQRT14PD \_\_m512d \_\_mm512\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m512d a);

VRSQRT14PD \_\_m256d \_\_mm256\_rsqrt14\_pd( \_\_m256d a);

VRSQRT14PD \_\_m256d \_\_mm256\_mask\_rsqrt14\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VRSQRT14PD \_\_m256d \_\_mm256\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m256d a);

VRSQRT14PD \_\_m128d \_\_mm128\_rsqrt14\_pd( \_\_m128d a);

VRSQRT14PD \_\_m128d \_\_mm128\_mask\_rsqrt14\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VRSQRT14PD \_\_m128d \_\_mm128\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.



## VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F38.W1 4F /r<br>VRSQRT14SD xmm1 {k1}{z},<br>xmm2, xmm3/m64 | T1S       | V/V                          | AVX512F                  | Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_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.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

### Operation

#### VRSQRT14SD (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← APPROXIMATE(1.0/ SQRT(SRC2[63:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**Table 5-27. VRSQRT14SD Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +0              |                          |

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SD \_\_m128d \_\_mm\_rsqrt14\_sd( \_\_m128d a, \_\_m128d b);

VRSQRT14SD \_\_m128d \_\_mm\_mask\_rsqrt14\_sd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VRSQRT14SD \_\_m128d \_\_mm\_maskz\_rsqrt14\_sd( \_\_mmask8d m, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.

## VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 4E /r<br>VRSQRT14PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask. |
| EVEX.256.66.0F38.W0 4E /r<br>VRSQRT14PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst | FV        | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask. |
| EVEX.512.66.0F38.W0 4E /r<br>VRSQRT14PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst | FV        | V/V                          | AVX512F                  | Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ .

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_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.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****VRSQRT14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[31:0]));

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Table 5-28. VRSQRT14PS Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +0              |                          |

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PS \_\_m512 \_\_mm512\_rsqrt14\_ps( \_\_m512 a);

VRSQRT14PS \_\_m512 \_\_mm512\_mask\_rsqrt14\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VRSQRT14PS \_\_m512 \_\_mm512\_maskz\_rsqrt14\_ps( \_\_mmask16 k, \_\_m512 a);

VRSQRT14PS \_\_m256 \_\_mm256\_rsqrt14\_ps( \_\_m256 a);

VRSQRT14PS \_\_m256 \_\_mm256\_mask\_rsqrt14\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRSQRT14PS \_\_m256 \_\_mm256\_maskz\_rsqrt14\_ps( \_\_mmask8 k, \_\_m256 a);

VRSQRT14PS \_\_m128 \_\_mm\_rsqrt14\_ps( \_\_m128 a);

VRSQRT14PS \_\_m128 \_\_mm\_mask\_rsqrt14\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VRSQRT14PS \_\_m128 \_\_mm\_maskz\_rsqrt14\_ps( \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F38.W0 4F /r<br>VRSQRT14SS xmm1 {k1}{z},<br>xmm2, xmm3/m32 | T1S       | V/V                          | AVX512F                  | Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (w) | VEX.vvvv  | ModRM:r/m (r) | NA        |

### Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $\infty$ , zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. 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.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

### Operation

#### VRSQRT14SS (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← APPROXIMATE(1.0/ SQRT(SRC2[31:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

Table 5-29. VRSQRT14SS Special Cases

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +0              |                          |

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SS \_\_m128 \_\_mm\_rsqrt14\_ss( \_\_m128 a, \_\_m128 b);

VRSQRT14SS \_\_m128 \_\_mm\_mask\_rsqrt14\_ss( \_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VRSQRT14SS \_\_m128 \_\_mm\_maskz\_rsqrt14\_ss( \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.

## VSCALEFPD—Scale Packed Float64 Values With Float64 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.128.66.0F38.W1 2C /r<br>VSCALEFPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1. |
| EVEX.NDS.256.66.0F38.W1 2C /r<br>VSCALEFPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1. |
| EVEX.NDS.512.66.0F38.W1 2C /r<br>VSCALEFPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

$$\text{zmm1} := \text{zmm2} * 2^{\text{floor}(\text{zmm3})}$$

Floor(zmm3) means maximum integer value  $\leq$  zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-31.

**Table 5-30. VSCALEFPD/SD/PS/SS Special Cases**

|      |             | Src2       |                       |                     |                | Set IE                   |
|------|-------------|------------|-----------------------|---------------------|----------------|--------------------------|
|      |             | $\pm$ NaN  | +Inf                  | -Inf                | 0/Denorm/Norm  |                          |
| Src1 | $\pm$ QNaN  | QNaN(Src1) | +INF                  | +0                  | QNaN(Src1)     | IF either source is SNAN |
|      | $\pm$ SNaN  | QNaN(Src1) | QNaN(Src1)            | QNaN(Src1)          | QNaN(Src1)     | YES                      |
|      | $\pm$ Inf   | QNaN(Src2) | Src1                  | QNaN_Indefinite     | Src1           | IF Src2 is SNAN or -INF  |
|      | $\pm$ 0     | QNaN(Src2) | QNaN_Indefinite       | Src1                | Src1           | IF Src2 is SNAN or +INF  |
|      | Denorm/Norm | QNaN(Src2) | $\pm$ INF (Src1 sign) | $\pm$ 0 (Src1 sign) | Compute Result | IF Src2 is SNAN          |

Table 5-31. Additional VSCALEFPD/SD Special Cases

| Special Case                    | Returned value  | Faults    |
|---------------------------------|---|-----------|
| $ \text{result}  < 2^{-1074}$   | $\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)                    | Underflow |
| $ \text{result}  \geq 2^{1024}$ | $\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign) | Overflow  |

**Operation**

```

SCALE(SRC1, SRC2)
{
  TMP_SRC2 ← SRC2
  TMP_SRC1 ← SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 64 bits floating-point value */
  DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
VSCALEFPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
    ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPD __m512d __mm512_scaleg_round_pd(__m512d a, __m512d b, int);
VSCALEFPD __m512d __mm512_mask_scaleg_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m512d __mm512_maskz_scaleg_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m256d __mm256_scaleg_round_pd(__m256d a, __m256d b, int);
VSCALEFPD __m256d __mm256_mask_scaleg_round_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m256d __mm256_maskz_scaleg_round_pd(__mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m128d __mm_scaleg_round_pd(__m128d a, __m128d b, int);
VSCALEFPD __m128d __mm_mask_scaleg_round_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFPD __m128d __mm_maskz_scaleg_round_pd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.

## VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F38.W1 2D /r<br>VSCALEFSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Scale the scalar double-precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

$$xmm1 := xmm2 * 2^{\text{floor}(xmm3)}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-31.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

**VSCALEFSD (EVEX encoded version)**

```

IF (EVEX.b= 1) and SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
  THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63:0] ← 0
    FI
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFSD __m128d __mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d __mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d __mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E3.

## VSCALEFPS—Scale Packed Float32 Values With Float32 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.128.66.0F38.W0 2C /r<br>VSCALEFPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.                |
| EVEX.NDS.256.66.0F38.W0 2C /r<br>VSCALEFPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.                |
| EVEX.NDS.512.66.0F38.W0 2C /r<br>VSCALEFPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| FV    | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

#### Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{\text{floor}(zmm3)}$$

Floor(zmm3) means maximum integer value  $\leq zmm3$ .

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-32.

**Table 5-32. Additional VSCALEFPS/SS Special Cases**

| Special Case                   | Returned value  | Faults    |
|--------------------------------|---|-----------|
| $ \text{result}  < 2^{-149}$   | $\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)                    | Underflow |
| $ \text{result}  \geq 2^{128}$ | $\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign) | Overflow  |

**Operation**

```

SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

```

**VSCALEFPS (EVEX encoded versions)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
            ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPS __m512 __mm512_scalef_round_ps(__m512 a, __m512 b, int);
VSCALEFPS __m512 __mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m512 __mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m256 __mm256_scalef_round_ps(__m256 a, __m256 b, int);
VSCALEFPS __m256 __mm256_mask_scalef_round_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m256 __mm256_maskz_scalef_round_ps(__mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m128 __mm_scalef_round_ps(__m128 a, __m128 b, int);
VSCALEFPS __m128 __mm_mask_scalef_round_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFPS __m128 __mm_maskz_scalef_round_ps(__mmask8 k, __m128 a, __m128 b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.

## VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F38.W0 2D /r<br>VSCALEFSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S       | V/V                         | AVX512F                  | Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-32.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

```
VSCALEFSS (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
```

```

        ELSE                                ; zeroing-masking
            DEST[31:0] ← 0
    FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

```

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

#### Other Exceptions

See Exceptions Type E3.

## VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

| Opcode/<br>Instruction                                       | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 A2 /vsib<br>VSCATTERDPS vm32x {k1}, xmm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A2 /vsib<br>VSCATTERDPS vm32y {k1}, ymm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A2 /vsib<br>VSCATTERDPS vm32z {k1}, zmm1 | T1S   | V/V                          | AVX512F                  | Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A2 /vsib<br>VSCATTERDPD vm32x {k1}, xmm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A2 /vsib<br>VSCATTERDPD vm32x {k1}, ymm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A2 /vsib<br>VSCATTERDPD vm32y {k1}, zmm1 | T1S   | V/V                          | AVX512F                  | Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1. |
| EVEX.128.66.0F38.W0 A3 /vsib<br>VSCATTERQPS vm64x {k1}, xmm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A3 /vsib<br>VSCATTERQPS vm64y {k1}, xmm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A3 /vsib<br>VSCATTERQPS vm64z {k1}, ymm1 | T1S   | V/V                          | AVX512F                  | Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A3 /vsib<br>VSCATTERQPD vm64x {k1}, xmm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A3 /vsib<br>VSCATTERQPD vm64y {k1}, ymm1 | T1S   | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A3 /vsib<br>VSCATTERQPD vm64z {k1}, zmm1 | T1S   | V/V                          | AVX512F                  | Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1   | Operand 2     | Operand 3 | Operand 4 |
|-------|---|---------------|-----------|-----------|
| T1S   | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | ModRM:reg (r) | NA        | NA        |

### Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE\_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corre-



sponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (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 register (k1) are partially updated. If any traps or interrupts are pending from already scattered 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.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- 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 zmm 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 scattered 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.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

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.

The instruction will #UD fault if the k0 mask register is specified

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VSCATTERDPS (EVEX encoded versions)**

(KL, VL)= (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP] ←
      SRC[i+31:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERDPD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ←
      SRC[i+63:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERQPS (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ←
      SRC[i+31:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERQPD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] ←
      SRC[i+63:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCATTERDPD void __mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void __mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void __mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void __mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void __mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void __mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void __mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void __mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);
VSCATTERDPD void __mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void __mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERDPS void __mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void __mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void __mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void __mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void __mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void __mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERDPD void __mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void __mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void __mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void __mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void __mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void __mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void __mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void __mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);

```

**SIMD Floating-Point Exceptions**

Invalid, Overflow, Underflow, Precision, Denormal

**Other Exceptions**

See Exceptions Type E12.

## SUBPD—Subtract Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| 66 0F 5C /r<br>SUBPD xmm1, xmm2/m128   | RM        | V/V                          | SSE2                     | Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.                            |
| VEX.NDS.128.66.0F.WIG 5C /r<br>VSUBPD xmm1,xmm2, xmm3/m128                         | RVM       | V/V                          | AVX                      | Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.                            |
| VEX.NDS.256.66.0F.WIG 5C /r<br>VSUBPD ymm1, ymm2, ymm3/m256                        | RVM       | V/V                          | AVX                      | Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.                            |
| EVEX.NDS.128.66.0F.W1 5C /r<br>VSUBPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Subtract packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1. |
| EVEX.NDS.256.66.0F.W1 5C /r<br>VSUBPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Subtract packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1. |
| EVEX.NDS.512.66.0F.W1 5C /r<br>VSUBPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | FV        | V/V                          | AVX512F                  | Subtract packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD subtract of the two, four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VSUBPD (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VSUBPD (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1)

THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0];

ELSE EST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VSUBPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]

DEST[127:64] ← SRC1[127:64] - SRC2[127:64]

DEST[191:128] ← SRC1[191:128] - SRC2[191:128]

DEST[255:192] ← SRC1[255:192] - SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VSUBPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64] - SRC2[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**SUBPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] - SRC[63:0]  
 DEST[127:64] ← DEST[127:64] - SRC[127:64]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSUBPD \_\_m512d \_\_mm512\_sub\_pd (\_\_m512d a, \_\_m512d b);  
 VSUBPD \_\_m512d \_\_mm512\_mask\_sub\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VSUBPD \_\_m512d \_\_mm512\_maskz\_sub\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VSUBPD \_\_m512d \_\_mm512\_sub\_round\_pd (\_\_m512d a, \_\_m512d b, int);  
 VSUBPD \_\_m512d \_\_mm512\_mask\_sub\_round\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 VSUBPD \_\_m512d \_\_mm512\_maskz\_sub\_round\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 VSUBPD \_\_m256d \_\_mm256\_sub\_pd (\_\_m256d a, \_\_m256d b);  
 VSUBPD \_\_m256d \_\_mm256\_mask\_sub\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VSUBPD \_\_m256d \_\_mm256\_maskz\_sub\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 SUBPD \_\_m128d \_\_mm\_sub\_pd (\_\_m128d a, \_\_m128d b);  
 VSUBPD \_\_m128d \_\_mm\_mask\_sub\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VSUBPD \_\_m128d \_\_mm\_maskz\_sub\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.  
 EVEX-encoded instructions, see Exceptions Type E2.

## SUBPS—Subtract Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| OF 5C /r<br>SUBPS xmm1, xmm2/m128   | RM        | V/V                          | SSE                      | Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.                             |
| VEX.NDS.128.OF.WIG 5C /r<br>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.OF.WIG 5C /r<br>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.                            |
| EVEX.NDS.128.OF.WO 5C /r<br>VSUBPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1. |
| EVEX.NDS.256.OF.WO 5C /r<br>VSUBPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | FV        | V/V                          | AVX512VL<br>AVX512F      | Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1. |
| EVEX.NDS.512.OF.WO 5C /r<br>VSUBPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | FV        | V/V                          | AVX512F                  | Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VSUBPS (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VSUBPS (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1)

THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VSUBPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]

DEST[63:32] ← SRC1[63:32] - SRC2[63:32]

DEST[95:64] ← SRC1[95:64] - SRC2[95:64]

DEST[127:96] ← SRC1[127:96] - SRC2[127:96]

DEST[159:128] ← SRC1[159:128] - SRC2[159:128]

DEST[191:160] ← SRC1[191:160] - SRC2[191:160]

DEST[223:192] ← SRC1[223:192] - SRC2[223:192]

DEST[255:224] ← SRC1[255:224] - SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0



**VSUBPS (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAX\_VL}-1:128] \leftarrow 0$

**SUBPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAX\_VL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VSUBPS __m512 __mm512_sub_ps (__m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_sub_round_ps (__m512 a, __m512 b, int);`  
`VSUBPS __m512 __mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VSUBPS __m512 __mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);`  
`VSUBPS __m256 __mm256_sub_ps (__m256 a, __m256 b);`  
`VSUBPS __m256 __mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VSUBPS __m256 __mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);`  
`SUBPS __m128 __mm_sub_ps (__m128 a, __m128 b);`  
`VSUBPS __m128 __mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VSUBPS __m128 __mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## SUBSD—Subtract Scalar Double-Precision Floating-Point Value

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| F2 0F 5C /r<br>SUBSD xmm1, xmm2/m64                                       | RM         | V/V                          | SSE2                     | Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.                    |
| VEX.NDS.128.F2.0F.WIG 5C /r<br>VSUBSD xmm1, xmm2, xmm3/m64                | RVM        | V/V                          | AVX                      | Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.                    |
| EVEX.NDS.LIG.F2.0F.W1 5C /r<br>VSUBSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | T1S        | V/V                          | AVX512F                  | Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSUBSD (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VSUBSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**SUBSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.  
 EVEX-encoded instructions, see Exceptions Type E3.

**SUBSS—Subtract Scalar Single-Precision Floating-Point Value**

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| F3 0F 5C /r<br>SUBSS xmm1, xmm2/m32                                       | RM         | V/V                          | SSE                      | Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.                    |
| VEX.NDS.128.F3.0F.WIG 5C /r<br>VSUBSS xmm1, xmm2, xmm3/m32                | RVM        | V/V                          | AVX                      | Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.                    |
| EVEX.NDS.LIG.F3.0F.W0 5C /r<br>VSUBSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | T1S        | V/V                          | AVX512F                  | Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1. |

**Instruction Operand Encoding**

| Op/En | 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        |
| T1S   | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSUBSS (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VSUBSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**SUBSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

| Opcode/<br>Instruction                                  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| 66 0F 2E /r<br>UCOMISD xmm1, xmm2/m64                   | RM        | V/V                          | SSE2                     | Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly. |
| VEX.128.66.0F.WIG 2E /r<br>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. |
| EVEX.LIG.66.0F.W1 2E /r<br>VUCOMISD xmm1, xmm2/m64{sae} | T1S       | V/V                          | AVX512F                  | Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (r) | ModRM:r/m (r) | NA        | NA        |
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Performs an unordered compare of the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid numeric exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)UCOMISD (all versions)

```
RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
  UNORDERED: ZF,PF,CF ← 111;
```

```
  GREATER_THAN: ZF,PF,CF ← 000;
```

```
  LESS_THAN: ZF,PF,CF ← 001;
```

```
  EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VUCOMISD int \_\_mm\_comi\_round\_sd(\_\_m128d a, \_\_m128d b, int imm, int sae);  
 UCOMISD int \_\_mm\_ucomieq\_sd(\_\_m128d a, \_\_m128d b)  
 UCOMISD int \_\_mm\_ucomilt\_sd(\_\_m128d a, \_\_m128d b)  
 UCOMISD int \_\_mm\_ucomile\_sd(\_\_m128d a, \_\_m128d b)  
 UCOMISD int \_\_mm\_ucomigt\_sd(\_\_m128d a, \_\_m128d b)  
 UCOMISD int \_\_mm\_ucomige\_sd(\_\_m128d a, \_\_m128d b)  
 UCOMISD int \_\_mm\_ucomineq\_sd(\_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

Invalid (if SNaN operands), Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

**UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS**

| Opcode/<br>Instruction                               | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| OF 2E /r<br>UCOMISS xmm1, xmm2/m32                   | RM        | V/V                          | SSE                      | Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |
| VEX.128.OF.WIG 2E /r<br>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. |
| EVEX.LIG.OF.WO 2E /r<br>VUCOMISS xmm1, xmm2/m32{sae} | T1S       | V/V                          | AVX512F                  | Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (r) | ModRM:r/m (r) | NA        | NA        |
| T1S   | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

**Description**

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#1) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****(V)UCOMISS (all versions)**

```

RESULT ← UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF ← 111;
  GREATER_THAN: ZF,PF,CF ← 000;
  LESS_THAN: ZF,PF,CF ← 001;
  EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }

```



**Intel C/C++ Compiler Intrinsic Equivalent**

VUCOMISS int \_mm\_comi\_round\_ss(\_\_m128 a, \_\_m128 b, int imm, int sae);  
 UCOMISS int \_mm\_ucomieq\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomilt\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomile\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomigt\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomige\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomineq\_ss(\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN Operands), Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 15 /r<br>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<br>VUNPCKHPD xmm1,xmm2,<br>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<br>VUNPCKHPD ymm1,ymm2,<br>ymm3/m256                  | RVM        | V/V                          | AVX                      | Unpacks and Interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.                                 |
| EVEX.NDS.128.66.0F.W1 15 /r<br>VUNPCKHPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1. |
| EVEX.NDS.256.66.0F.W1 15 /r<br>VUNPCKHPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1. |
| EVEX.NDS.512.66.0F.W1 15 /r<br>VUNPCKHPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Unpacks and Interleaves double-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKHPD (EVEX encoded versions when SRC2 is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128

    TMP\_DEST[63:0] ← SRC1[127:64]

    TMP\_DEST[127:64] ← SRC2[127:64]

FI;

IF VL >= 256

    TMP\_DEST[191:128] ← SRC1[255:192]

    TMP\_DEST[255:192] ← SRC2[255:192]

FI;

IF VL >= 512

    TMP\_DEST[319:256] ← SRC1[383:320]

    TMP\_DEST[383:320] ← SRC2[383:320]

    TMP\_DEST[447:384] ← SRC1[511:448]

    TMP\_DEST[511:448] ← SRC2[511:448]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VUNPCKHPD (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1)

THEN TMP\_SRC2[i+63:i] ← SRC2[63:0]

ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i]

FI;

ENDFOR;

IF VL &gt;= 128

TMP\_DEST[63:0] ← SRC1[127:64]

TMP\_DEST[127:64] ← TMP\_SRC2[127:64]

FI;

IF VL &gt;= 256

TMP\_DEST[191:128] ← SRC1[255:192]

TMP\_DEST[255:192] ← TMP\_SRC2[255:192]

FI;

IF VL &gt;= 512

TMP\_DEST[319:256] ← SRC1[383:320]

TMP\_DEST[383:320] ← TMP\_SRC2[383:320]

TMP\_DEST[447:384] ← SRC1[511:448]

TMP\_DEST[511:448] ← TMP\_SRC2[511:448]

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VUNPCKHPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[127:64]

DEST[127:64] ← SRC2[127:64]

DEST[191:128] ← SRC1[255:192]

DEST[255:192] ← SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VUNPCKHPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[127:64]

DEST[127:64] ← SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

**UNPCKHPD (128-bit Legacy SSE version)**

DEST[63:0] ← SRC1[127:64]

DEST[127:64] ← SRC2[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKHPD \_\_m512d \_\_mm512\_unpackhi\_pd(\_\_m512d a, \_\_m512d b);  
 VUNPCKHPD \_\_m512d \_\_mm512\_mask\_unpackhi\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKHPD \_\_m512d \_\_mm512\_maskz\_unpackhi\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKHPD \_\_m256d \_\_mm256\_unpackhi\_pd(\_\_m256d a, \_\_m256d b);  
 VUNPCKHPD \_\_m256d \_\_mm256\_mask\_unpackhi\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VUNPCKHPD \_\_m256d \_\_mm256\_maskz\_unpackhi\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 UNPCKHPD \_\_m128d \_\_mm\_unpackhi\_pd(\_\_m128d a, \_\_m128d b);  
 VUNPCKHPD \_\_m128d \_\_mm\_mask\_unpackhi\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VUNPCKHPD \_\_m128d \_\_mm\_maskz\_unpackhi\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| OF 15 /r<br>UNPCKHPS xmm1, xmm2/m128   | RM         | V/V                          | SSE                      | Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.  |
| VEX.NDS.128.OF.WIG 15 /r<br>VUNPCKHPS xmm1, xmm2,<br>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.OF.WIG 15 /r<br>VUNPCKHPS ymm1, ymm2,<br>ymm3/m256                 | RVM        | V/V                          | AVX                      | Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.  |
| EVEX.NDS.128.OF.W0 15 /r<br>VUNPCKHPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1. |
| EVEX.NDS.256.OF.W0 15 /r<br>VUNPCKHPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1. |
| EVEX.NDS.512.OF.W0 15 /r<br>VUNPCKHPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination 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.

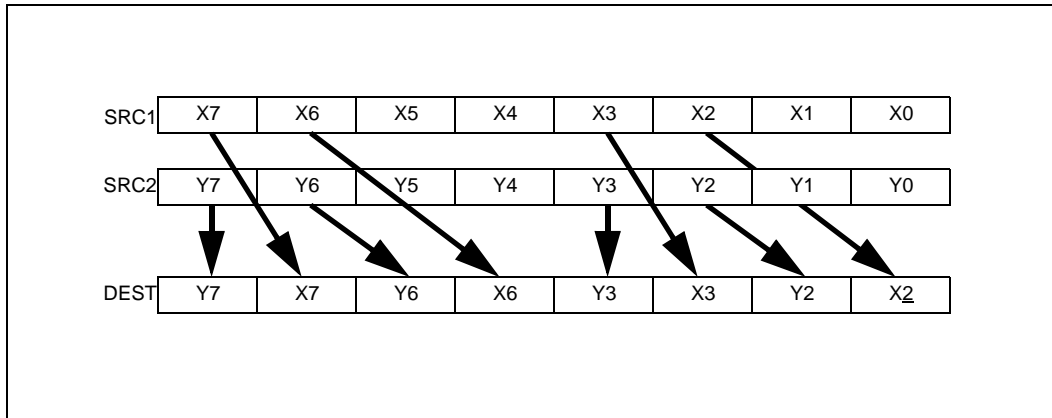


Figure 5-43. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKHPS (EVEX encoded version when SRC2 is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL >= 128

```
TMP_DEST[31:0] ← SRC1[95:64]
TMP_DEST[63:32] ← SRC2[95:64]
TMP_DEST[95:64] ← SRC1[127:96]
TMP_DEST[127:96] ← SRC2[127:96]
```

FI;

IF VL >= 256

```
TMP_DEST[159:128] ← SRC1[223:192]
TMP_DEST[191:160] ← SRC2[223:192]
TMP_DEST[223:192] ← SRC1[255:224]
TMP_DEST[255:224] ← SRC2[255:224]
```

FI;

IF VL >= 512

```
TMP_DEST[287:256] ← SRC1[351:320]
TMP_DEST[319:288] ← SRC2[351:320]
TMP_DEST[351:320] ← SRC1[383:352]
TMP_DEST[383:352] ← SRC2[383:352]
TMP_DEST[415:384] ← SRC1[479:448]
TMP_DEST[447:416] ← SRC2[479:448]
TMP_DEST[479:448] ← SRC1[511:480]
TMP_DEST[511:480] ← SRC2[511:480]
```

FI;

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKHPS (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] ← SRC1[95:64]
  TMP_DEST[63:32] ← TMP_SRC2[95:64]
  TMP_DEST[95:64] ← SRC1[127:96]
  TMP_DEST[127:96] ← TMP_SRC2[127:96]
FI;
IF VL >= 256
  TMP_DEST[159:128] ← SRC1[223:192]
  TMP_DEST[191:160] ← TMP_SRC2[223:192]
  TMP_DEST[223:192] ← SRC1[255:224]
  TMP_DEST[255:224] ← TMP_SRC2[255:224]
FI;
IF VL >= 512
  TMP_DEST[287:256] ← SRC1[351:320]
  TMP_DEST[319:288] ← TMP_SRC2[351:320]
  TMP_DEST[351:320] ← SRC1[383:352]
  TMP_DEST[383:352] ← TMP_SRC2[383:352]
  TMP_DEST[415:384] ← SRC1[479:448]
  TMP_DEST[447:416] ← TMP_SRC2[479:448]
  TMP_DEST[479:448] ← SRC1[511:480]
  TMP_DEST[511:480] ← TMP_SRC2[511:480]
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```



FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKHPS (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]
DEST[MAX_VL-1:256] ← 0

```

**VUNPCKHPS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**UNPCKHPS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps( __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps( __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| 66 0F 14 /r<br>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<br>VUNPCKLPD xmm1,xmm2,<br>xmm3/m128                  | RVM        | V/V                          | AVX                      | Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.                                  |
| VEX.NDS.256.66.0F.WIG 14 /r<br>VUNPCKLPD ymm1,ymm2,<br>ymm3/m256                  | RVM        | V/V                          | AVX                      | Unpacks and Interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.                                  |
| EVEX.NDS.128.66.0F.W1 14 /r<br>VUNPCKLPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1. |
| EVEX.NDS.256.66.0F.W1 14 /r<br>VUNPCKLPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1. |
| EVEX.NDS.512.66.0F.W1 14 /r<br>VUNPCKLPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512F                  | Unpacks and Interleaves double-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKLPD (EVEX encoded versions when SRC2 is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128

    TMP\_DEST[63:0] ← SRC1[63:0]

    TMP\_DEST[127:64] ← SRC2[63:0]

FI;

IF VL >= 256

    TMP\_DEST[191:128] ← SRC1[191:128]

    TMP\_DEST[255:192] ← SRC2[191:128]

FI;

IF VL >= 512

    TMP\_DEST[319:256] ← SRC1[319:256]

    TMP\_DEST[383:320] ← SRC2[319:256]

    TMP\_DEST[447:384] ← SRC1[447:384]

    TMP\_DEST[511:448] ← SRC2[447:384]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VUNPCKLPD (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[63:0] ← SRC1[63:0]
  TMP_DEST[127:64] ← TMP_SRC2[63:0]
FI;
IF VL >= 256
  TMP_DEST[191:128] ← SRC1[191:128]
  TMP_DEST[255:192] ← TMP_SRC2[191:128]
FI;
IF VL >= 512
  TMP_DEST[319:256] ← SRC1[319:256]
  TMP_DEST[383:320] ← TMP_SRC2[319:256]
  TMP_DEST[447:384] ← SRC1[447:384]
  TMP_DEST[511:448] ← TMP_SRC2[447:384]
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKLPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]
DEST[MAX_VL-1:256] ← 0

```

**VUNPCKLPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] ← 0

```

**UNPCKLPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKLPD \_\_m512d \_mm512\_unpacklo\_pd( \_\_m512d a, \_\_m512d b);  
 VUNPCKLPD \_\_m512d \_mm512\_mask\_unpacklo\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKLPD \_\_m512d \_mm512\_maskz\_unpacklo\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKLPD \_\_m256d \_mm256\_unpacklo\_pd(\_\_m256d a, \_\_m256d b)  
 VUNPCKLPD \_\_m256d \_mm256\_mask\_unpacklo\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VUNPCKLPD \_\_m256d \_mm256\_maskz\_unpacklo\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 UNPCKLPD \_\_m128d \_mm\_unpacklo\_pd(\_\_m128d a, \_\_m128d b)  
 VUNPCKLPD \_\_m128d \_mm\_mask\_unpacklo\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VUNPCKLPD \_\_m128d \_mm\_maskz\_unpacklo\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| OF 14 /r<br>UNPCKLPS xmm1, xmm2/m128   | RM         | V/V                          | SSE                      | Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.   |
| VEX.NDS.128.OF.WIG 14 /r<br>VUNPCKLPS xmm1, xmm2,<br>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.OF.WIG 14 /r<br>VUNPCKLPS<br>ymm1, ymm2, ymm3/m256                 | RVM        | V/V                          | AVX                      | Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.   |
| EVEX.NDS.128.OF.W0 14 /r<br>VUNPCKLPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.          |
| EVEX.NDS.256.OF.W0 14 /r<br>VUNPCKLPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512F      | Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.          |
| EVEX.NDS.512.OF.W0 14 /r<br>VUNPCKLPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512F                  | Unpacks and Interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1. |

### Instruction Operand Encoding

| Op/En | 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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

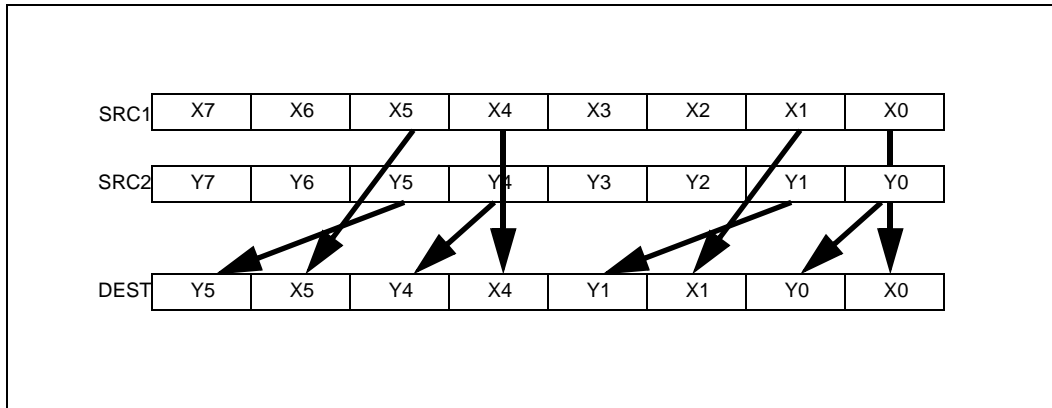


Figure 5-44. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL >= 128

```
TMP_DEST[31:0] ← SRC1[31:0]
TMP_DEST[63:32] ← SRC2[31:0]
TMP_DEST[95:64] ← SRC1[63:32]
TMP_DEST[127:96] ← SRC2[63:32]
```

FI;

IF VL >= 256

```
TMP_DEST[159:128] ← SRC1[159:128]
TMP_DEST[191:160] ← SRC2[159:128]
TMP_DEST[223:192] ← SRC1[191:160]
TMP_DEST[255:224] ← SRC2[191:160]
```

FI;

IF VL >= 512

```
TMP_DEST[287:256] ← SRC1[287:256]
TMP_DEST[319:288] ← SRC2[287:256]
TMP_DEST[351:320] ← SRC1[319:288]
TMP_DEST[383:352] ← SRC2[319:288]
TMP_DEST[415:384] ← SRC1[415:384]
TMP_DEST[447:416] ← SRC2[415:384]
TMP_DEST[479:448] ← SRC1[447:416]
TMP_DEST[511:480] ← SRC2[447:416]
```

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

```

IF k1[j] OR *no writemask*
  THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
      ELSE *zeroing-masking*       ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKLPS (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 31

IF (EVEX.b = 1)

THEN TMP\_SRC2[j+31:i] ← SRC2[31:0]

ELSE TMP\_SRC2[j+31:i] ← SRC2[j+31:i]

FI;

ENDFOR;

IF VL &gt;= 128

TMP\_DEST[31:0] ← SRC1[31:0]

TMP\_DEST[63:32] ← TMP\_SRC2[31:0]

TMP\_DEST[95:64] ← SRC1[63:32]

TMP\_DEST[127:96] ← TMP\_SRC2[63:32]

FI;

IF VL &gt;= 256

TMP\_DEST[159:128] ← SRC1[159:128]

TMP\_DEST[191:160] ← TMP\_SRC2[159:128]

TMP\_DEST[223:192] ← SRC1[191:160]

TMP\_DEST[255:224] ← TMP\_SRC2[191:160]

FI;

IF VL &gt;= 512

TMP\_DEST[287:256] ← SRC1[287:256]

TMP\_DEST[319:288] ← TMP\_SRC2[287:256]

TMP\_DEST[351:320] ← SRC1[319:288]

TMP\_DEST[383:352] ← TMP\_SRC2[319:288]

TMP\_DEST[415:384] ← SRC1[415:384]

TMP\_DEST[447:416] ← TMP\_SRC2[415:384]

TMP\_DEST[479:448] ← SRC1[447:416]

TMP\_DEST[511:480] ← TMP\_SRC2[447:416]

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[j+31:i] ← TMP\_DEST[j+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+31:i] ← 0

FI

FI;



ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

**UNPCKLPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[159:128] ← SRC1[159:128]  
 DEST[191:160] ← SRC2[159:128]  
 DEST[223:192] ← SRC1[191:160]  
 DEST[255:224] ← SRC2[191:160]  
 DEST[MAX\_VL-1:256] ← 0

**VUNPCKLPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[MAX\_VL-1:128] ← 0

**UNPCKLPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKLPS \_\_m512 \_\_mm512\_unpacklo\_ps(\_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m512 \_\_mm512\_mask\_unpacklo\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m512 \_\_mm512\_maskz\_unpacklo\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m256 \_\_mm256\_unpacklo\_ps (\_\_m256 a, \_\_m256 b);  
 VUNPCKLPS \_\_m256 \_\_mm256\_mask\_unpacklo\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VUNPCKLPS \_\_m256 \_\_mm256\_maskz\_unpacklo\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 UNPCKLPS \_\_m128 \_\_mm\_unpacklo\_ps (\_\_m128 a, \_\_m128 b);  
 VUNPCKLPS \_\_m128 \_\_mm\_mask\_unpacklo\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VUNPCKLPS \_\_m128 \_\_mm\_maskz\_unpacklo\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

**XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values**

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| 66 0F 57/r<br>XORPD xmm1, xmm2/m128  | RM         | V/V                          | SSE2                     | Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.                                  |
| VEX.NDS.128.66.0F.WIG 57 /r<br>VXORPD xmm1,xmm2,<br>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<br>VXORPD ymm1, ymm2,<br>ymm3/m256                 | RVM        | V/V                          | AVX                      | Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.                                  |
| EVEX.NDS.128.66.0F.W1 57 /r<br>VXORPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1. |
| EVEX.NDS.256.66.0F.W1 57 /r<br>VXORPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1. |
| EVEX.NDS.512.66.0F.W1 57 /r<br>VXORPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | FV         | V/V                          | AVX512DQ                 | Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1. |

**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        |
| FV    | ModRM:reg (w)    | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

**Description**

Performs a bitwise logical XOR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0];

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 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]

DEST[MAX\_VL-1:256] ← 0

**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[MAX\_VL-1:128] ← 0

**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[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VXORPD \_\_m512d \_\_mm512\_xor\_pd (\_\_m512d a, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_mask\_xor\_pd (\_\_m512d a, \_\_mmask8 m, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m512d a);

VXORPD \_\_m256d \_\_mm256\_xor\_pd (\_\_m256d a, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_mask\_xor\_pd (\_\_m256d a, \_\_mmask8 m, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m256d a);

XORPD \_\_m128d \_\_mm\_xor\_pd (\_\_m128d a, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_mask\_xor\_pd (\_\_m128d a, \_\_mmask8 m, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

## XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| OF 57 /r<br>XORPS xmm1, xmm2/m128   | RM         | V/V                          | SSE                      | Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.                                  |
| VEX.NDS.128.OF.WIG 57 /r<br>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.OF.WIG 57 /r<br>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.                                  |
| EVEX.NDS.128.OF.W0 57 /r<br>VXORPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1. |
| EVEX.NDS.256.OF.W0 57 /r<br>VXORPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | FV         | V/V                          | AVX512VL<br>AVX512DQ     | Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1. |
| EVEX.NDS.512.OF.W0 57 /r<br>VXORPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | FV         | V/V                          | AVX512DQ                 | Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1. |

### 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      | ModRM:r/m (r) | NA        |
| FV    | ModRM:reg (w)    | EVEX.vvvv     | ModRM:r/m (r) | NA        |

### Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 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].

DEST[MAX\_VL-1:256] ← 0

**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[MAX\_VL-1:128] ← 0

**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[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VXORPS \_\_m512 \_\_mm512\_xor\_ps (\_\_m512 a, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_mask\_xor\_ps (\_\_m512 a, \_\_mmask16 m, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_maskz\_xor\_ps (\_\_mmask16 m, \_\_m512 a);

VXORPS \_\_m256 \_\_mm256\_xor\_ps (\_\_m256 a, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_mask\_xor\_ps (\_\_m256 a, \_\_mmask8 m, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m256 a);

XORPS \_\_m128 \_\_mm\_xor\_ps (\_\_m128 a, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_mask\_xor\_ps (\_\_m128 a, \_\_mmask8 m, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m128 a);

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

This page was  
intentionally left  
blank.



## CHAPTER 6

# INSTRUCTION SET REFERENCE - OPMASK

Instructions for data transfer between opmask registers and between opmask/general-purpose registers are described in this chapter using the same notations and conventions listed in *Section 5.1* and *Section 5.1.5.1*.

## 6.1 MASK INSTRUCTIONS

### KADDW/KADDB/KADDQ/KADD—ADD Two Masks

| Opcode/<br>Instruction                    | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-------|------------------------------|--------------------------|--|
| VEX.L1.0F.W0 4A /r<br>KADDW k1, k2, k3    | RVR   | V/V                          | AVX512DQ                 | Add 16 bits masks in k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W0 4A /r<br>KADDB k1, k2, k3 | RVR   | V/V                          | AVX512DQ                 | Add 8 bits masks in k2 and k3 and place result in k1.  |
| VEX.L1.0F.W1 4A /r<br>KADDQ k1, k2, k3    | RVR   | V/V                          | AVX512BW                 | Add 64 bits masks in k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W1 4A /r<br>KADD k1, k2, k3  | RVR   | V/V                          | AVX512BW                 | Add 32 bits masks in k2 and k3 and place result in k1. |

#### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vvv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

#### Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

#### Operation

##### KADDW

DEST[15:0] ← SRC1[15:0] + SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

##### KADDB

DEST[7:0] ← SRC1[7:0] + SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

##### KADDQ

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

##### KADD

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

| Opcode/<br>Instruction                     | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-------|------------------------------|--------------------------|---|
| VEX.NDS.L1.0F.W0 41 /r<br>KANDW k1, k2, k3 | RVR   | V/V                          | AVX512F                  | Bitwise AND 16 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W0 41 /r<br>KANDB k1, k2, k3  | RVR   | V/V                          | AVX512DQ                 | Bitwise AND 8 bits masks k2 and k3 and place result in k1.  |
| VEX.L1.0F.W1 41 /r<br>KANDQ k1, k2, k3     | RVR   | V/V                          | AVX512BW                 | Bitwise AND 64 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W1 41 /r<br>KANDD k1, k2, k3  | RVR   | V/V                          | AVX512BW                 | Bitwise AND 32 bits masks k2 and k3 and place result in k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vsv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

#### Description

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

#### Operation

##### KANDW

DEST[15:0] ← SRC1[15:0] BITWISE AND SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

##### KANDB

DEST[7:0] ← SRC1[7:0] BITWISE AND SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

##### KANDQ

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

##### KANDD

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

KANDW `__mmask16 _mm512_kand(__mmask16 a, __mmask16 b);`

#### Flags Affected

None

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type K20.

**KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks**

| Opcode/<br>Instruction                      | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-------|------------------------------|--------------------------|---|
| VEX.NDS.L1.0F.W0 42 /r<br>KANDNW k1, k2, k3 | RVR   | V/V                          | AVX512F                  | Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W0 42 /r<br>KANDNB k1, k2, k3  | RVR   | V/V                          | AVX512DQ                 | Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.  |
| VEX.L1.0F.W1 42 /r<br>KANDNQ k1, k2, k3     | RVR   | V/V                          | AVX512BW                 | Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W1 42 /r<br>KANDND k1, k2, k3  | RVR   | V/V                          | AVX512BW                 | Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vvv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

**Description**

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

**Operation****KANDNW**

DEST[15:0] ← (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

**KANDNB**

DEST[7:0] ← (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

**KANDNQ**

DEST[63:0] ← (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

**KANDND**

DEST[31:0] ← (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KANDNW \_\_mmask16 \_\_mm512\_kandn(\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KMOVW/KMOVB/KMOVQ/KMOVD—Move from and to Mask Registers

| Opcode/<br>Instruction                    | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-------|------------------------------|--------------------------|---|
| VEX.L0.0F.W0 90 /r<br>KMOVW k1, k2/m16    | RM    | V/V                          | AVX512F                  | Move 16 bits mask from k2/m16 and store the result in k1. |
| VEX.L0.66.0F.W0 90 /r<br>KMOVB k1, k2/m8  | RM    | V/V                          | AVX512DQ                 | Move 8 bits mask from k2/m8 and store the result in k1.   |
| VEX.L0.0F.W1 90 /r<br>KMOVQ k1, k2/m64    | RM    | V/V                          | AVX512BW                 | Move 64 bits mask from k2/m64 and store the result in k1. |
| VEX.L0.66.0F.W1 90 /r<br>KMOVD k1, k2/m32 | RM    | V/V                          | AVX512BW                 | Move 32 bits mask from k2/m32 and store the result in k1. |
| VEX.L0.0F.W0 91 /r<br>KMOVW m16, k1       | MR    | V/V                          | AVX512F                  | Move 16 bits mask from k1 and store the result in m16.    |
| VEX.L0.66.0F.W0 91 /r<br>KMOVB m8, k1     | MR    | V/V                          | AVX512DQ                 | Move 8 bits mask from k1 and store the result in m8.      |
| VEX.L0.0F.W1 91 /r<br>KMOVQ m64, k1       | MR    | V/V                          | AVX512BW                 | Move 64 bits mask from k1 and store the result in m64.    |
| VEX.L0.66.0F.W1 91 /r<br>KMOVD m32, k1    | MR    | V/V                          | AVX512BW                 | Move 32 bits mask from k1 and store the result in m32.    |
| VEX.L0.0F.W0 92 /r<br>KMOVW k1, r32       | RR    | V/V                          | AVX512F                  | Move 16 bits mask from r32 to k1.                         |
| VEX.L0.66.0F.W0 92 /r<br>KMOVB k1, r32    | RR    | V/V                          | AVX512DQ                 | Move 8 bits mask from r32 to k1.                          |
| VEX.L0.F2.0F.W1 92 /r<br>KMOVQ k1, r64    | RR    | V/I                          | AVX512BW                 | Move 64 bits mask from r64 to k1.                         |
| VEX.L0.F2.0F.W0 92 /r<br>KMOVD k1, r32    | RR    | V/V                          | AVX512BW                 | Move 32 bits mask from r32 to k1.                         |
| VEX.L0.0F.W0 93 /r<br>KMOVW r32, k1       | RR    | V/V                          | AVX512F                  | Move 16 bits mask from k1 to r32.                         |
| VEX.L0.66.0F.W0 93 /r<br>KMOVB r32, k1    | RR    | V/V                          | AVX512DQ                 | Move 8 bits mask from k1 to r32.                          |
| VEX.L0.F2.0F.W1 93 /r<br>KMOVQ r64, k1    | RR    | V/I                          | AVX512BW                 | Move 64 bits mask from k1 to r64.                         |
| VEX.L0.F2.0F.W0 93 /r<br>KMOVD r32, k1    | RR    | V/V                          | AVX512BW                 | Move 32 bits mask from k1 to r32.                         |

### Instruction Operand Encoding

| Op/En | Operand 1                                  | Operand 2                              |
|-------|--|--|
| RM    | ModRM:reg (w)                              | ModRM:r/m (r)                          |
| MR    | ModRM:r/m (w, ModRM:[7:6] must not be 11b) | ModRM:reg (r)                          |
| RR    | ModRM:reg (w)                              | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

### Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX\_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the

default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that REX.W cannot be used to modify the size of the general-purpose destination.

### Operation

#### **KMOVW**

IF \*destination is a memory location\*

$DEST[15:0] \leftarrow SRC[15:0]$

IF \*destination is a mask register or a GPR \*

$DEST \leftarrow ZeroExtension(SRC[15:0])$

#### **KMOVB**

IF \*destination is a memory location\*

$DEST[7:0] \leftarrow SRC[7:0]$

IF \*destination is a mask register or a GPR \*

$DEST \leftarrow ZeroExtension(SRC[7:0])$

#### **KMOVQ**

IF \*destination is a memory location or a GPR\*

$DEST[63:0] \leftarrow SRC[63:0]$

IF \*destination is a mask register\*

$DEST \leftarrow ZeroExtension(SRC[63:0])$

#### **KMOVD**

IF \*destination is a memory location\*

$DEST[31:0] \leftarrow SRC[31:0]$

IF \*destination is a mask register or a GPR \*

$DEST \leftarrow ZeroExtension(SRC[31:0])$

### Intel C/C++ Compiler Intrinsic Equivalent

`KMOVW __mmask16 _mm512_kmov(__mmask16 a);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Instructions with RR operand encoding See Exceptions Type K20.

Instructions with RM or MR operand encoding See Exceptions Type K21.

## KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

| Opcode/<br>Instruction                           | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-------|------------------------------|--------------------------|---|
| VEX.NDS.L1.66.OF.W0 4B /r<br>KUNPCKBW k1, k2, k3 | RVR   | V/V                          | AVX512F                  | Unpack and interleave 8 bits masks in k2 and k3 and write word result in k1.      |
| VEX.NDS.L1.0F.W0 4B /r<br>KUNPCKWD k1, k2, k3    | RVR   | V/V                          | AVX512BW                 | Unpack and interleave 16 bits in k2 and k3 and write double-word result in k1.    |
| VEX.NDS.L1.0F.W1 4B /r<br>KUNPCKDQ k1, k2, k3    | RVR   | V/V                          | AVX512BW                 | Unpack and interleave 32 bits masks in k2 and k3 and write quadword result in k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vvv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

#### Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

#### Operation

##### KUNPCKBW

```
DEST[7:0] ← SRC2[7:0]
DEST[15:8] ← SRC1[7:0]
DEST[MAX_KL-1:16] ← 0
```

##### KUNPCKWD

```
DEST[15:0] ← SRC2[15:0]
DEST[31:16] ← SRC1[15:0]
DEST[MAX_KL-1:32] ← 0
```

##### KUNPCKDQ

```
DEST[31:0] ← SRC2[31:0]
DEST[63:32] ← SRC1[31:0]
DEST[MAX_KL-1:64] ← 0
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```
KUNPCKBW __mmask16 __mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 __mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 __mm512_kunpackw(__mmask32 a, __mmask32 b);
```

#### Flags Affected

None

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type K20.

**KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register**

| Opcode/<br>Instruction                | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature Flag | Description                     |
|---------------------------------------|-------|------------------------------|-----------------------|---------------------------------|
| VEX.LO.0F.W0 44 /r<br>KNOTW k1, k2    | RR    | V/V                          | AVX512F               | Bitwise NOT of 16 bits mask k2. |
| VEX.LO.66.0F.W0 44 /r<br>KNOTB k1, k2 | RR    | V/V                          | AVX512DQ              | Bitwise NOT of 8 bits mask k2.  |
| VEX.LO.0F.W1 44 /r<br>KNOTQ k1, k2    | RR    | V/V                          | AVX512BW              | Bitwise NOT of 64 bits mask k2. |
| VEX.LO.66.0F.W1 44 /r<br>KNOTD k1, k2 | RR    | V/V                          | AVX512BW              | Bitwise NOT of 32 bits mask k2. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2                              |
|-------|---------------|--|
| RR    | ModRM:reg (w) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

**Description**

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

**Operation****KNOTW**

DEST[15:0] ← BITWISE NOT SRC[15:0]

DEST[MAX\_KL-1:16] ← 0

**KNOTB**

DEST[7:0] ← BITWISE NOT SRC[7:0]

DEST[MAX\_KL-1:8] ← 0

**KNOTQ**

DEST[63:0] ← BITWISE NOT SRC[63:0]

DEST[MAX\_KL-1:64] ← 0

**KNOTD**

DEST[31:0] ← BITWISE NOT SRC[31:0]

DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KNOTW `__mmask16 _mm512_knot(__mmask16 a);`

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.



## KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

| Opcode/<br>Instruction                    | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-------|------------------------------|--------------------------|--|
| VEX.NDS.L1.0F.W0 45 /r<br>KORW k1, k2, k3 | RVR   | V/V                          | AVX512F                  | Bitwise OR 16 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W0 45 /r<br>KORB k1, k2, k3  | RVR   | V/V                          | AVX512DQ                 | Bitwise OR 8 bits masks k2 and k3 and place result in k1.  |
| VEX.L1.0F.W1 45 /r<br>KORQ k1, k2, k3     | RVR   | V/V                          | AVX512BW                 | Bitwise OR 64 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W1 45 /r<br>KORD k1, k2, k3  | RVR   | V/V                          | AVX512BW                 | Bitwise OR 32 bits masks k2 and k3 and place result in k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vvv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

#### Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

#### Operation

##### KORW

DEST[15:0] ← SRC1[15:0] BITWISE OR SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

##### KORB

DEST[7:0] ← SRC1[7:0] BITWISE OR SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

##### KORQ

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

##### KORD

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

KORW \_\_mmask16 \_mm512\_kor(\_\_mmask16 a, \_\_mmask16 b);

#### Flags Affected

None

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type K20.

**KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks And Set Flags**

| Opcode/<br>Instruction                   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.L0.0F.W0 98 /r<br>KORTESTW k1, k2    | RR        | V/V                          | AVX512F                  | Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly. |
| VEX.L0.66.0F.W0 98 /r<br>KORTESTB k1, k2 | RR        | V/V                          | AVX512DQ                 | Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.  |
| VEX.L0.0F.W1 98 /r<br>KORTESTQ k1, k2    | RR        | V/V                          | AVX512BW                 | Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly. |
| VEX.L0.66.0F.W1 98 /r<br>KORTESTD k1, k2 | RR        | V/V                          | AVX512BW                 | Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2                              |
|-------|---------------|--|
| RR    | ModRM:reg (w) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

**Description**

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

**Operation****KORTESTW**

TMP[15:0] ← DEST[15:0] BITWISE OR SRC[15:0]

IF(TMP[15:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

FI;

IF(TMP[15:0]=FFFFh)

THEN CF ← 1

ELSE CF ← 0

FI;

**KORTESTB**

TMP[7:0] ← DEST[7:0] BITWISE OR SRC[7:0]

IF(TMP[7:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

FI;

IF(TMP[7:0]=FFh)

THEN CF ← 1

ELSE CF ← 0

FI;

**KORTESTQ**

TMP[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]

IF(TMP[63:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

```

FI;
IF(TMP[63:0]==FFFFFFFF_FFFFFFFFh)
  THEN CF ← 1
  ELSE CF ← 0
FI;

```

**KORTESTD**

```

TMP[31:0] ← DEST[31:0] BITWISE OR SRC[31:0]
IF(TMP[31:0]=0)
  THEN ZF ← 1
  ELSE ZF ← 0
FI;
IF(TMP[31:0]=FFFFFFFFh)
  THEN CF ← 1
  ELSE CF ← 0
FI;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);
```

**Flags Affected**

The ZF flag is set if the result of OR-ing both sources is all 0s.

The CF flag is set if the result of OR-ing both sources is all 1s.

The OF, SF, AF, and PF flags are set to 0.

**Other Exceptions**

See Exceptions Type K20.

**KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers**

| Opcode/<br>Instruction                           | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-------|------------------------------|--------------------------|---|
| VEX.L0.66.0F3A.W1 32 /r<br>KSHIFTLW k1, k2, imm8 | RRI   | V/V                          | AVX512F                  | Shift left 16 bits in k2 by immediate and write result in k1. |
| VEX.L0.66.0F3A.W0 32 /r<br>KSHIFTLB k1, k2, imm8 | RRI   | V/V                          | AVX512DQ                 | Shift left 8 bits in k2 by immediate and write result in k1.  |
| VEX.L0.66.0F3A.W1 33 /r<br>KSHIFTLQ k1, k2, imm8 | RRI   | V/V                          | AVX512BW                 | Shift left 64 bits in k2 by immediate and write result in k1. |
| VEX.L0.66.0F3A.W0 33 /r<br>KSHIFTLD k1, k2, imm8 | RRI   | V/V                          | AVX512BW                 | Shift left 32 bits in k2 by immediate and write result in k1. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2                              | Operand 3 |
|-------|---------------|--|-----------|
| RRI   | ModRM:reg (w) | ModRM:r/m (r, ModRM:[7:6] must be 11b) | Imm8      |

**Description**

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

**Operation****KSHIFTLW**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 15
    THEN DEST[15:0] ← SRC1[15:0] << COUNT;
FI;
```

**KSHIFTLB**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 7
    THEN DEST[7:0] ← SRC1[7:0] << COUNT;
FI;
```

**KSHIFTLQ**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 63
    THEN DEST[63:0] ← SRC1[63:0] << COUNT;
FI;
```

**KSHIFTLD**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 31
    THEN DEST[31:0] ← SRC1[31:0] << COUNT;
```

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTLW when needed.

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

**KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers**

| Opcode/<br>Instruction                           | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| VEX.LO.66.0F3A.W1 30 /r<br>KSHIFTRW k1, k2, imm8 | RRI   | V/V                          | AVX512F                  | Shift right 16 bits in k2 by immediate and write result in k1. |
| VEX.LO.66.0F3A.W0 30 /r<br>KSHIFTRB k1, k2, imm8 | RRI   | V/V                          | AVX512DQ                 | Shift right 8 bits in k2 by immediate and write result in k1.  |
| VEX.LO.66.0F3A.W1 31 /r<br>KSHIFTRQ k1, k2, imm8 | RRI   | V/V                          | AVX512BW                 | Shift right 64 bits in k2 by immediate and write result in k1. |
| VEX.LO.66.0F3A.W0 31 /r<br>KSHIFTRD k1, k2, imm8 | RRI   | V/V                          | AVX512BW                 | Shift right 32 bits in k2 by immediate and write result in k1. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2                              | Operand 3 |
|-------|---------------|--|-----------|
| RRI   | ModRM:reg (w) | ModRM:r/m (r, ModRM:[7:6] must be 11b) | Imm8      |

**Description**

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

**Operation****KSHIFTRW**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 15
    THEN DEST[15:0] ← SRC1[15:0] >> COUNT;
FI;
```

**KSHIFTRB**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 7
    THEN DEST[7:0] ← SRC1[7:0] >> COUNT;
FI;
```

**KSHIFTRQ**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 63
    THEN DEST[63:0] ← SRC1[63:0] >> COUNT;
FI;
```

**KSHIFTRD**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 31
    THEN DEST[31:0] ← SRC1[31:0] >> COUNT;
```

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTRW when needed.

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

**KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks**

| Opcode/<br>Instruction                      | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-------|------------------------------|--------------------------|--|
| VEX.NDS.L1.0F.W0 46 /r<br>KXNORW k1, k2, k3 | RVR   | V/V                          | AVX512F                  | Bitwise XNOR 16 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W0 46 /r<br>KXNORB k1, k2, k3  | RVR   | V/V                          | AVX512DQ                 | Bitwise XNOR 8 bits masks k2 and k3 and place result in k1.  |
| VEX.L1.0F.W1 46 /r<br>KXNORQ k1, k2, k3     | RVR   | V/V                          | AVX512BW                 | Bitwise XNOR 64 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W1 46 /r<br>KXNORD k1, k2, k3  | RVR   | V/V                          | AVX512BW                 | Bitwise XNOR 32 bits masks k2 and k3 and place result in k1. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vvv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

**Description**

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

**Operation****KXNORW**

DEST[15:0] ← NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])  
DEST[MAX\_KL-1:16] ← 0

**KXNORB**

DEST[7:0] ← NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])  
DEST[MAX\_KL-1:8] ← 0

**KXNORQ**

DEST[63:0] ← NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])  
DEST[MAX\_KL-1:64] ← 0

**KXNORD**

DEST[31:0] ← NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KXNORW \_\_mmask16 \_\_mm512\_kxnor(\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.



**KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags**

| Opcode/<br>Instruction                 | Op<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|----------|------------------------------|--------------------------|--|
| VEX.L0.0F.W0 99 /r<br>KTESTW k1, k2    | RR       | V/V                          | AVX512DQ                 | Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources. |
| VEX.L0.66.0F.W0 99 /r<br>KTESTB k1, k2 | RR       | V/V                          | AVX512DQ                 | Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.  |
| VEX.L0.0F.W1 99 /r<br>KTESTQ k1, k2    | RR       | V/V                          | AVX512BW                 | Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources. |
| VEX.L0.66.0F.W1 99 /r<br>KTESTD k1, k2 | RR       | V/V                          | AVX512BW                 | Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand2                               |
|-------|---------------|--|
| RR    | ModRM:reg (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

**Description**

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****KTESTW**

TEMP[15:0] ← SRC2[15:0] AND SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN ZF ← 1;

ELSE ZF ← 0;

FI;

TEMP[15:0] ← SRC2[15:0] AND NOT SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN CF ← 1;

ELSE CF ← 0;

FI;

AF ← OF ← PF ← SF ← 0;

**KTESTB**

TEMP[7:0] ← SRC2[7:0] AND SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN ZF ← 1;

ELSE ZF ← 0;

FI;

TEMP[7:0] ← SRC2[7:0] AND NOT SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN CF ← 1;

ELSE CF  $\leftarrow$  0;  
FI;  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**KTESTQ**

TEMP[63:0]  $\leftarrow$  SRC2[63:0] AND SRC1[63:0]  
IF (TEMP[63:0] = 0)  
    THEN ZF  $\leftarrow$  1;  
    ELSE ZF  $\leftarrow$  0;  
FI;  
TEMP[63:0]  $\leftarrow$  SRC2[63:0] AND NOT SRC1[63:0]  
IF (TEMP[63:0] = 0)  
    THEN CF  $\leftarrow$  1;  
    ELSE CF  $\leftarrow$  0;  
FI;  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**KTESTD**

TEMP[31:0]  $\leftarrow$  SRC2[31:0] AND SRC1[31:0]  
IF (TEMP[31:0] = 0)  
    THEN ZF  $\leftarrow$  1;  
    ELSE ZF  $\leftarrow$  0;  
FI;  
TEMP[31:0]  $\leftarrow$  SRC2[31:0] AND NOT SRC1[31:0]  
IF (TEMP[31:0] = 0)  
    THEN CF  $\leftarrow$  1;  
    ELSE CF  $\leftarrow$  0;  
FI;  
AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**Intel C/C++ Compiler Intrinsic Equivalent**

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

| Opcode/<br>Instruction                     | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-------|------------------------------|--------------------------|---|
| VEX.NDS.L1.0F.W0 47 /r<br>KXORW k1, k2, k3 | RVR   | V/V                          | AVX512F                  | Bitwise XOR 16 bits masks k2 and k3 and place result in k1. |
| VEX.L1.66.0F.W0 47 /r<br>KXORB k1, k2, k3  | RVR   | V/V                          | AVX512DQ                 | Bitwise XOR 8 bits masks k2 and k3 and place result in k1   |
| VEX.L1.0F.W1 47 /r<br>KXORQ k1, k2, k3     | RVR   | V/V                          | AVX512BW                 | Bitwise XOR 64 bits masks k2 and k3 and place result in k1  |
| VEX.L1.66.0F.W1 47 /r<br>KXORD k1, k2, k3  | RVR   | V/V                          | AVX512BW                 | Bitwise XOR 32 bits masks k2 and k3 and place result in k1  |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3                              |
|-------|---------------|--------------|--|
| RVR   | ModRM:reg (w) | VEX.1vvv (r) | ModRM:r/m (r, ModRM:[7:6] must be 11b) |

#### Description

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

#### Operation

##### KXORW

DEST[15:0] ← SRC1[15:0] BITWISE XOR SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

##### KXORB

DEST[7:0] ← SRC1[7:0] BITWISE XOR SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

##### KXORQ

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

##### KXORD

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

KXORW `__mmask16 _mm512_xor(__mmask16 a, __mmask16 b);`

#### Flags Affected

None

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type K20.

This page was  
intentionally left  
blank.

# CHAPTER 7

## ADDITIONAL 512-BIT INSTRUCTION EXTENSIONS

This chapter describes additional 512-bit instruction extensions to accelerate specific application domain, such as certain transcendental mathematic computations, or specific prefetch operations. These instructions operate on 512-bit ZMM, support opmask registers, are encoded using the same EVEX prefix encoding format, and require the same operating system support as AVX-512 Foundation instructions. The application programming model described in Chapter 2 also applies. Instructions described in this chapter follow the general documentation convention established in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A and 2B*. Additional notations and conventions adopted in this document are listed in *Section 5.1*. *Section 5.1.5.1* covers supplemental information that applies to a specific subset of instructions.

The instructions VEXP2PD/VEXP2PS/VRCP28xx/VRSQRT28xx, also known as Intel AVX-512 Exponential and Reciprocal instructions, provide building blocks for accelerating certain transcendental math computations. The instructions VGATHERPF0xxx/VGATHERPF1xxx/VSCATTERPF0xxx/VSCATTERPF1xxx, Intel AVX-512 Prefetch instructions, can be useful for reducing memory operation latency exposure that involve gather/scatter instructions.

### 7.1 DETECTION OF 512-BIT INSTRUCTION EXTENSIONS

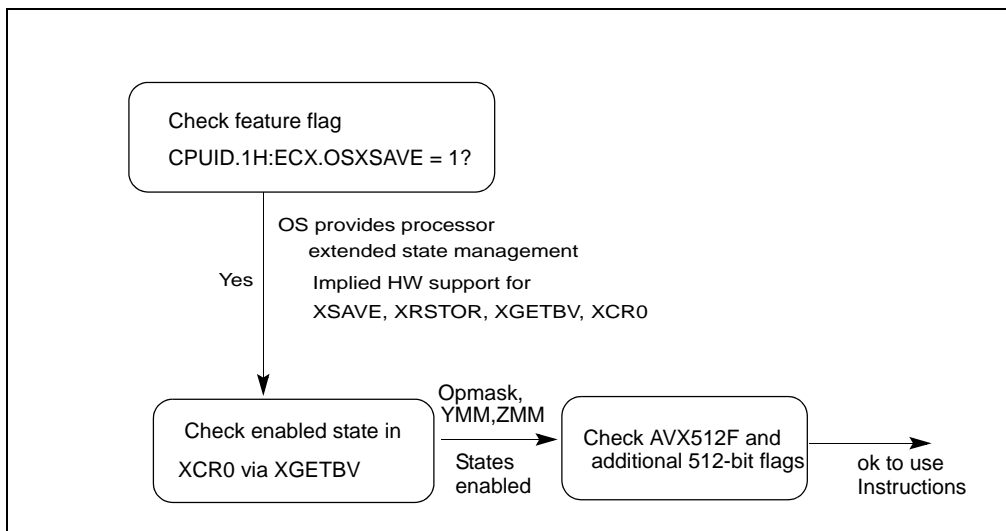
Processor support of the Intel AVX-512 Exponential and Reciprocal instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512ER[bit 27] = 1, the collection of VEXP2PD/VEXP2PS/VRCP28xx/VRSQRT28xx instructions are supported

Processor support of the Intel AVX-512 Prefetch instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512PF[bit 26] = 1, a collection of VGATHERPF0xxx/VGATHERPF1xxx/VSCATTERPF0xxx/VSCATTERPF1xxx instructions are supported.

Detection of 512-bit instructions operating on ZMM states and opmask registers, outside of AVX-512 Foundation, need to follow the general procedural flow in Figure 7-1.



**Figure 7-1. Procedural Flow of Application Detection of 512-bit Instructions**

Procedural Flow of Application Detection of other 512-bit extensions:

Prior to using the Intel AVX-512 Exponential and Reciprocal instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor’s support for

ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

- 1) Detect `CPUID.1:ECX.OSXSAVE[bit 27] = 1` (XGETBV enabled for application use)
- 2) Execute XGETBV and verify that `XCR0[7:5] = '111b'` (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that `XCR0[2:1] = '11b'` (XMM state and YMM state are enabled by OS).
- 3) Verify both `CPUID.0x7.0:EBX.AVX512F[bit 16] = 1`, and `CPUID.0x7.0:EBX.AVX512ER[bit 27] = 1`.

Prior to using the Intel AVX-512 Prefetch instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

- 1) Detect `CPUID.1:ECX.OSXSAVE[bit 27] = 1` (XGETBV enabled for application use)
- 2) Execute XGETBV and verify that `XCR0[7:5] = '111b'` (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that `XCR0[2:1] = '11b'` (XMM state and YMM state are enabled by OS).
- 3) Verify both `CPUID.0x7.0:EBX.AVX512F[bit 16] = 1`, and `CPUID.0x7.0:EBX.AVX512PF[bit 26] = 1`.

## 7.2 INSTRUCTION SET REFERENCE

## VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W1 C8 /r<br>VEXP2PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst {sae} | FV        | V/V                          | AVX512ER                 | Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 | Operand 4 |
|-------|------------------|---------------|-----------|-----------|
| FV    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        | NA        |

### Description

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than  $2^{-23}$  of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VEXP2PD

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] ← EXP2\_23\_DP(SRC[63:0])

      ELSE DEST[i+63:i] ← EXP2\_23\_DP(SRC[i+63:i])

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

    FI;

  FI;

ENDFOR;

**Table 7-1. Special Values Behavior**

| Source Input     | Result           | Comments                |
|------------------|------------------|-------------------------|
| NaN              | QNaN(src)        | If (SRC = SNaN) then #I |
| +∞               | +∞               |                         |
| +/-0             | 1.0f             | <i>Exact result</i>     |
| -∞               | +0.0f            |                         |
| Integral value N | 2 <sup>(N)</sup> | <i>Exact result</i>     |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VEXP2PD __m512d __mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d __mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d __mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.



## VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 C8 /r<br>VEXP2PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst {sae} | FV        | V/V                          | AVX512ER                 | Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 | Operand 4 |
|-------|------------------|---------------|-----------|-----------|
| FV    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        | NA        |

### Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than  $2^{-23}$  of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VEXP2PS

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i] ← EXP2\_23\_SP(SRC[31:0])

      ELSE DEST[i+31:i] ← EXP2\_23\_SP(SRC[i+31:i])

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

    FI;

  FI;

ENDFOR;

**Table 7-2. Special Values Behavior**

| Source Input     | Result           | Comments                |
|------------------|------------------|-------------------------|
| NaN              | QNaN(src)        | If (SRC = SNaN) then #I |
| +∞               | +∞               |                         |
| +/-0             | 1.0f             | <i>Exact result</i>     |
| -∞               | +0.0f            |                         |
| Integral value N | 2 <sup>(N)</sup> | <i>Exact result</i>     |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VEXP2PS __m512 __mm512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 __mm512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 __mm512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.

## VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W1 CA /r<br>VRCP28PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst {sae} | FV        | V/V                          | AVX512ER                 | Computes the approximate reciprocals ( $< 2^{-28}$ relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VRCP28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← RCP_28_DP(1.0/SRC[63:0]);
      ELSE DEST[i+63:i] ← RCP_28_DP(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;

```

**Table 7-3. VRCP28PD Special Cases**

| Input value              | Result value | Comments   |
|--------------------------|--------------|--|
| NAN                      | QNAN(input)  | If (SRC = SNaN) then #I                          |
| $0 \leq X < 2^{-1022}$   | INF          | Positive input denormal or zero; #Z              |
| $-2^{-1022} < X \leq -0$ | -INF         | Negative input denormal or zero; #Z              |
| $X > 2^{1022}$           | +0.0f        |  |
| $X < -2^{1022}$          | -0.0f        |  |
| $X = +\infty$            | +0.0f        |  |
| $X = -\infty$            | -0.0f        |  |
| $X = 2^{-n}$             | $2^n$        | Exact result (unless input/output is a denormal) |
| $X = -2^{-n}$            | $-2^n$       | Exact result (unless input/output is a denormal) |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRCP28PD __m512d _mm512_rcp28_round_pd( __m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd( __mmask8 m, __m512d b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2<sup>-28</sup> Relative Error

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.NDS.LIG.66.0F38.W1 CB /r<br>VRCP28SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64 {sae} | T1S       | V/V                          | AVX512ER                 | Computes the approximate reciprocal (< 2 <sup>-28</sup> relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper 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 |
|-------|---------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2<sup>-28</sup> of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VRCP28SD ((EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[63: 0] ← RCP_28_DP(1.0/SRC2[63: 0]);
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[63: 0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAX_VL-1:128] ← 0
    
```

**Table 7-4. VRCP28SD Special Cases**

| Input value              | Result value | Comments   |
|--------------------------|--------------|--|
| NAN                      | QNAN(input)  | If (SRC = SNaN) then #I                          |
| $0 \leq X < 2^{-1022}$   | INF          | Positive input denormal or zero; #Z              |
| $-2^{-1022} < X \leq -0$ | -INF         | Negative input denormal or zero; #Z              |
| $X > 2^{1022}$           | +0.0f        |  |
| $X < -2^{1022}$          | -0.0f        |  |
| $X = +\infty$            | +0.0f        |  |
| $X = -\infty$            | -0.0f        |  |
| $X = 2^{-n}$             | $2^n$        | Exact result (unless input/output is a denormal) |
| $X = -2^{-n}$            | $-2^n$       | Exact result (unless input/output is a denormal) |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 CA /r<br>VRCP28PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst {sae} | FV        | V/V                          | AVX512ER                 | Computes the approximate reciprocals ( $< 2^{-28}$ relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final results are rounded to  $< 2^{-23}$  relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VRCP28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] ← RCP_28_SP(1.0/SRC[31:0]);
      ELSE DEST[i+31:i] ← RCP_28_SP(1.0/SRC[i+31:i]);
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI;
  FI;
ENDFOR;

```

**Table 7-5. VRCP28PS Special Cases**

| Input value             | Result value | Comments   |
|-------------------------|--------------|--|
| NAN                     | QNAN(input)  | If (SRC = SNaN) then #I                          |
| $0 \leq X < 2^{-126}$   | INF          | Positive input denormal or zero; #Z              |
| $-2^{-126} < X \leq -0$ | -INF         | Negative input denormal or zero; #Z              |
| $X > 2^{126}$           | +0.0f        |  |
| $X < -2^{126}$          | -0.0f        |  |
| $X = +\infty$           | +0.0f        |  |
| $X = -\infty$           | -0.0f        |  |
| $X = 2^{-n}$            | $2^n$        | Exact result (unless input/output is a denormal) |
| $X = -2^{-n}$           | $-2^n$       | Exact result (unless input/output is a denormal) |

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28PS \_\_mm512\_rcp28\_round\_ps (\_\_m512 a, int sae);

VRCP28PS \_\_m512 \_\_mm512\_mask\_rcp28\_round\_ps(\_\_m512 s, \_\_mmask16 m, \_\_m512 a, int sae);

VRCP28PS \_\_m512 \_\_mm512\_maskz\_rcp28\_round\_ps(\_\_mmask16 m, \_\_m512 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.



## VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F38.WO CB /r<br>VRCP28SS xmm1 {k1}{z},<br>xmm2, xmm3/m32 {sae} | T1S       | V/V                          | AVX512ER                 | Computes the approximate reciprocal ( $< 2^{-28}$ relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3     | Operand 4 |
|-------|---------------|-----------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA        |

### Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final result is rounded to  $< 2^{-23}$  relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VRCP28SS ((EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[31: 0] ← RCP_28_SP(1.0/SRC2[31: 0]);
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[31: 0] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[31: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAX_VL-1:128] ← 0
    
```

**Table 7-6. VRCP28SS Special Cases**

| Input value             | Result value | Comments   |
|-------------------------|--------------|--|
| NAN                     | QNAN(input)  | If (SRC = SNaN) then #I                          |
| $0 \leq X < 2^{-126}$   | INF          | Positive input denormal or zero; #Z              |
| $-2^{-126} < X \leq -0$ | -INF         | Negative input denormal or zero; #Z              |
| $X > 2^{126}$           | +0.0f        |  |
| $X < -2^{126}$          | -0.0f        |  |
| $X = +\infty$           | +0.0f        |  |
| $X = -\infty$           | -0.0f        |  |
| $X = 2^{-n}$            | $2^n$        | Exact result (unless input/output is a denormal) |
| $X = -2^{-n}$           | $-2^n$       | Exact result (unless input/output is a denormal) |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRCP28SS __m128_mm_rcp28_round_ss ( __m128 a, __m128 b, int sae);
VRCP28SS __m128_mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128_mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W1 CC /r<br>VRSQRT28PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst {sae} | FV        | V/V                          | AVX512ER                 | Computes approximations to the Reciprocal square root ( $<2^{-28}$ relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VRSQRT28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i]  $\leftarrow$  (1.0/ SQRT(SRC[63:0]));

      ELSE DEST[i+63:i]  $\leftarrow$  (1.0/ SQRT(SRC[i+63:i]));

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

**Table 7-7. VRSQRT28PD Special Cases**

| Input value                   | Result value    | Comments                |
|-------------------------------|-----------------|-------------------------|
| NAN                           | QNAN(input)     | If (SRC = SNaN) then #I |
| $X = 2^{-2n}$                 | $2^n$           |                         |
| $X < 0$                       | QNAN_Indefinite | Including -INF          |
| $X = -0$ or negative denormal | -INF            | #Z                      |
| $X = +0$ or positive denormal | +INF            | #Z                      |
| $X = +INF$                    | +0              |                         |

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PD \_\_m512d \_\_mm512\_rsqrt28\_round\_pd(\_\_m512d a, int sae);

VRSQRT28PD \_\_m512d \_\_mm512\_mask\_rsqrt28\_round\_pd(\_\_m512d s, \_\_mmask8 m, \_\_m512d a, int sae);

VRSQRT28PD \_\_m512d \_\_mm512\_maskz\_rsqrt28\_round\_pd(\_\_mmask8 m, \_\_m512d a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F38.W1 CD /r<br>VRSQRT28SD xmm1 {k1}{z},<br>xmm2, xmm3/m64 {sae} | T1S       | V/V                          | AVX512ER                 | Computes approximate reciprocal square root ( $<2^{-28}$ relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1 with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than  $2^{-28}$  of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

### Operation

#### VRSQRT28SD (EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[63: 0] ← (1.0/ SQRT(SRC[63: 0]));
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[63: 0] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAX_VL-1:128] ← 0
    
```

**Table 7-8. VRSQRT28SD Special Cases**

| Input value                   | Result value    | Comments                |
|-------------------------------|-----------------|-------------------------|
| NAN                           | QNAN(input)     | If (SRC = SNaN) then #I |
| $X = 2^{-2n}$                 | $2^n$           |                         |
| $X < 0$                       | QNAN_Indefinite | Including -INF          |
| $X = -0$ or negative denormal | -INF            | #Z                      |
| $X = +0$ or positive denormal | +INF            | #Z                      |
| $X = +INF$                    | +0              |                         |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRSQRT28SD __m128d __mm_rsqrt28_round_sd(__m128d a, __m128d b, int sae);
VRSQRT28SD __m128d __mm_mask_rsqrt28_round_pd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRSQRT28SD __m128d __mm_maskz_rsqrt28_round_pd(__mmask8 m, __m128d a, __m128d b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 CC /r<br>VRSQRT28PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst {sae} | FV        | V/V                          | AVX512ER                 | Computes approximations to the Reciprocal square root (< $2^{-28}$ relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| FV    | ModRM:reg (w) | ModRM:r/m (r) | NA        | NA        |

### Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final results is rounded to  $< 2^{-23}$  relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VRSQRT28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)  
      THEN DEST[i+31:i] ← (1.0/ SQRT(SRC[31:0]));  
      ELSE DEST[i+31:i] ← (1.0/ SQRT(SRC[i+31:i]));

  FI;

  ELSE

    IF \*merging-masking\* ;merging-masking  
      THEN \*DEST[i+31:i] remains unchanged\*  
    ELSE ;zeroing-masking  
      DEST[i+31:i] ← 0

  FI;

  FI;

ENDFOR;

**Table 7-9. VRSQRT28PS Special Cases**

| Input value                   | Result value    | Comments                |
|-------------------------------|-----------------|-------------------------|
| NAN                           | QNAN(input)     | If (SRC = SNaN) then #I |
| $X = 2^{-2n}$                 | $2^n$           |                         |
| $X < 0$                       | QNAN_Indefinite | Including -INF          |
| $X = -0$ or negative denormal | -INF            | #Z                      |
| $X = +0$ or positive denormal | +INF            | #Z                      |
| $X = +INF$                    | +0              |                         |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRSQRT28PS __m512 __mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 __mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRSQRT28PS __m512 __mm512_maskz_rsqrt28_round_ps(__mmask16 m, __m512 a, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.



## VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.NDS.LIG.66.0F38.W0 CD /r<br>VRSQRT28SS xmm1 {k1}{z},<br>xmm2, xmm3/m32 {sae} | T1S       | V/V                          | AVX512ER                 | Computes approximate reciprocal square root ( $<2^{-28}$ relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1 with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|-----------|
| T1S   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA        |

### Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final result is rounded to  $< 2^{-23}$  relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

### Operation

#### VRSQRT28SS (EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[31: 0] ← (1.0/ SQRT(SRC[31: 0]));
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[31: 0] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[31: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAX_VL-1:128] ← 0
    
```

**Table 7-10. VRSQRT28SS Special Cases**

| Input value                   | Result value    | Comments                |
|-------------------------------|-----------------|-------------------------|
| NAN                           | QNAN(input)     | If (SRC = SNaN) then #I |
| $X = 2^{-2n}$                 | $2^n$           |                         |
| $X < 0$                       | QNAN_Indefinite | Including -INF          |
| $X = -0$ or negative denormal | -INF            | #Z                      |
| $X = +0$ or positive denormal | +INF            | #Z                      |
| $X = +INF$                    | +0              |                         |

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRSQRT28SS __m128 __mm_rsqrt28_round_ss(__m128 a, __m128 b, int sae);
VRSQRT28SS __m128 __mm512_mask_rsqrt28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRSQRT28SS __m128 __mm512_maskz_rsqrt28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VGATHERPFODPS/VGATHERPFOQPS/VGATHERPFODPD/VGATHERPFOQPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 C6 /1 /vsib<br>VGATHERPFODPS vm32z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint. |
| EVEX.512.66.0F38.W0 C7 /1 /vsib<br>VGATHERPFOQPS vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint. |
| EVEX.512.66.0F38.W1 C6 /1 /vsib<br>VGATHERPFODPD vm32y {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint. |
| EVEX.512.66.0F38.W1 C7 /1 /vsib<br>VGATHERPFOQPD vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint. |

### Instruction Operand Encoding

| Op/En | Operand 1   | Operand 2 | Operand 3 | Operand 4 |
|-------|---|-----------|-----------|-----------|
| T1S   | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        | NA        |

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

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.

## ADDITIONAL 512-BIT INSTRUCTION EXTENSIONS

### VGATHERPFODPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

### VGATHERPFODPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

### VGATHERPFOQPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

### VGATHERPFOQPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VGATHERPFOQPD void __mm512_mask_prefetch_j32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPFODPS void __mm512_mask_prefetch_j32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPFOQPD void __mm512_mask_prefetch_j64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPFOQPS void __mm512_mask_prefetch_j64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E12NP.

### VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 C6 /2 /vsib<br>VGATHERPF1DPS vm32z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint. |
| EVEX.512.66.0F38.W0 C7 /2 /vsib<br>VGATHERPF1QPS vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint. |
| EVEX.512.66.0F38.W1 C6 /2 /vsib<br>VGATHERPF1DPD vm32y {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint. |
| EVEX.512.66.0F38.W1 C7 /2 /vsib<br>VGATHERPF1QPD vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint. |

#### Instruction Operand Encoding

| Op/En | Operand 1   | Operand 2 | Operand 3 | Operand 4 |
|-------|---|-----------|-----------|-----------|
| T1S   | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        | NA        |

#### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

#### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

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.

## ADDITIONAL 512-BIT INSTRUCTION EXTENSIONS

### VGATHERPF1DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

### VGATHERPF1DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

### VGATHERPF1QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

### VGATHERPF1QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VGATHERPF1DPD void __mm512_mask_prefetch_j32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void __mm512_mask_prefetch_j32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void __mm512_mask_prefetch_j64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void __mm512_mask_prefetch_j64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E12NP.

## VSCATTERPFODPS/VSCATTERPFOQPS/VSCATTERPFODPD/VSCATTERPFOQPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 C6 /5 /vsib<br>VSCATTERPFODPS vm32z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write. |
| EVEX.512.66.0F38.W0 C7 /5 /vsib<br>VSCATTERPFOQPS vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write. |
| EVEX.512.66.0F38.W1 C6 /5 /vsib<br>VSCATTERPFODPD vm32y {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write. |
| EVEX.512.66.0F38.W1 C7 /5 /vsib<br>VSCATTERPFOQPD vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write. |

### Instruction Operand Encoding

| Op/En | Operand 1   | Operand 2 | Operand 3 | Operand 4 |
|-------|---|-----------|-----------|-----------|
| T1S   | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        | NA        |

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

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.

**VSCATTERPFODPS (EVEX encoded version)**

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

**VSCATTERPFODPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

**VSCATTERPFQPS (EVEX encoded version)**

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

**VSCATTERPFQPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VSCATTERPFODPD void __mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPFODPD void __mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPFODPS void __mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPFODPS void __mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPFQPD void __mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPFQPD void __mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPFQPS void __mm512_prefetch_i64scatter_ps(void * base, __m512i vdx, int scale, int hint);
VSCATTERPFQPS void __mm512_mask_prefetch_i64scatter_ps(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.



## VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 C6 /6 /vsib<br>VSCATTERPF1DPS vm32z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write. |
| EVEX.512.66.0F38.W0 C7 /6 /vsib<br>VSCATTERPF1QPS vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write. |
| EVEX.512.66.0F38.W1 C6 /6 /vsib<br>VSCATTERPF1DPD vm32y {k1} | T1S       | V/V                          | AVX512PF                 | Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write. |
| EVEX.512.66.0F38.W1 C7 /6 /vsib<br>VSCATTERPF1QPD vm64z {k1} | T1S       | V/V                          | AVX512PF                 | Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write. |

### Instruction Operand Encoding

| Op/En | Operand 1   | Operand 2 | Operand 3 | Operand 4 |
|-------|---|-----------|-----------|-----------|
| T1S   | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | NA        | NA        | NA        |

#### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

#### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

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.

**VSCATTERPF1DPS (EVEX encoded version)**

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

**VSCATTERPF1DPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

**VSCATTERPF1QPS (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

**VSCATTERPF1QPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VSCATTERPF1DPD void __mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void __mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void __mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPS void __mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void __mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void __mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void __mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void __mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.

# CHAPTER 8

## INTEL® SHA EXTENSIONS

### 8.1 OVERVIEW

This chapter describes a family of instruction extensions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants. The instruction syntax generally has two operands (in one case there is an implicit xmm0 register operand, in another a third immediate operand), where the first operand is an XMM register that provides the source as input and is the destination storing the result as well. The second source can be an XMM register or a 16-Byte aligned 128-bit memory location. In 64-bit mode, using a REX prefix in the form REX.R permits the instructions to access additional registers (XMM8-XMM15). The SHA extensions do not update any arithmetic flags and are valid in 32 and 64-bit modes. Exception behavior of the SHA extensions follows type 4 defined in Chapter 2 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

### 8.2 DETECTION OF INTEL SHA EXTENSIONS

A processor supports Intel SHA extensions if  $CPUID.(EAX=07H, ECX=0):EBX.SHA [bit 29] = 1$ . The SHA extensions require only XMM state support on operating systems, similar to SSE2 instructions.

#### 8.2.1 Common Transformations and Primitive Functions

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1 and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- $f_0()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.  

$$f_0(B,C,D) \leftarrow (B \text{ AND } C) \text{ XOR } ((\text{NOT}(B) \text{ AND } D))$$
- $f_1()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.  

$$f_1(B,C,D) \leftarrow B \text{ XOR } C \text{ XOR } D$$
- $f_2()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.  

$$f_2(B,C,D) \leftarrow (B \text{ AND } C) \text{ XOR } (B \text{ AND } D) \text{ XOR } (C \text{ AND } D)$$
- $f_3()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as  $f_1()$ .  

$$f_3(B,C,D) \leftarrow B \text{ XOR } C \text{ XOR } D$$
- $Ch()$ : A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).  

$$Ch(E,F,G) \leftarrow (E \text{ AND } F) \text{ XOR } ((\text{NOT } E) \text{ AND } G)$$
- $Maj()$ : A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).  

$$Maj(A,B,C) \leftarrow (A \text{ AND } B) \text{ XOR } (A \text{ AND } C) \text{ XOR } (B \text{ AND } C)$$

ROR is rotate right operation

$$(A \text{ ROR } N) \leftarrow A[N-1:0] \parallel A[\text{Width}-1:N]$$

ROL is rotate left operation

$$(A \text{ ROL } N) \leftarrow A \text{ ROR } (\text{Width}-N)$$

SHR is the right shift operation

$$(A \text{ SHR } N) \leftarrow \text{ZEROES}[N-1:0] \parallel A[\text{Width}-1:N]$$

- $\Sigma_0()$ : A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.  
 $\Sigma_0(A) \leftarrow (A \text{ ROR } 2) \text{ XOR } (A \text{ ROR } 13) \text{ XOR } (A \text{ ROR } 22)$
- $\Sigma_1()$ : A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.  
 $\Sigma_1(E) \leftarrow (E \text{ ROR } 6) \text{ XOR } (E \text{ ROR } 11) \text{ XOR } (E \text{ ROR } 25)$
- $\sigma_0()$ : A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.  
 $\sigma_0(W) \leftarrow (W \text{ ROR } 7) \text{ XOR } (W \text{ ROR } 18) \text{ XOR } (W \text{ SHR } 3)$
- $\sigma_1()$ : A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.  
 $\sigma_1(W) \leftarrow (W \text{ ROR } 17) \text{ XOR } (W \text{ ROR } 19) \text{ XOR } (W \text{ SHR } 10)$
- $K_i$ : SHA1 Constants dependent on immediate  $i$ .  
 $K_0 = 0x5A827999$   
 $K_1 = 0x6ED9EBA1$   
 $K_2 = 0X8F1BBCDC$   
 $K_3 = 0xCA62C1D6$

## 8.3 SHA EXTENSIONS REFERENCE

## SHA1RND54—Perform Four Rounds of SHA1 Operation

| Opcode/<br>Instruction                               | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-------|------------------------------|--------------------------|---|
| OF 3A CC /r ib<br>SHA1RND54 xmm1,<br>xmm2/m128, imm8 | RMI   | V/V                          | SHA                      | Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 |
|-------|------------------|---------------|-----------|
| RMI   | ModRM:reg (r, w) | ModRM:r/m (r) | Imm8      |

### Description

The SHA1RND54 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

### Operation

#### SHA1RND54

The function  $f()$  and Constant  $K$  are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN f() ← f0(), K ← K0;
ELSE IF ( imm8[1:0] = 1 )
    THEN f() ← f1(), K ← K1;
ELSE IF ( imm8[1:0] = 2 )
    THEN f() ← f2(), K ← K2;
ELSE IF ( imm8[1:0] = 3 )
    THEN f() ← f3(), K ← K3;
FI;
```

```
A ← SRC1[127:96];
B ← SRC1[95:64];
C ← SRC1[63:32];
D ← SRC1[31:0];
W0E ← SRC2[127:96];
W1 ← SRC2[95:64];
W2 ← SRC2[63:32];
W3 ← SRC2[31:0];
```

Round  $i = 0$  operation:

```
A_1 ← f (B, C, D) + (A ROL 5) +W0E +K;
B_1 ← A;
C_1 ← B ROL 30;
D_1 ← C;
E_1 ← D;
```

FOR  $i = 1$  to 3

```
A_(i+1) ← f (B_i, C_i, D_i) + (A_i ROL 5) +Wi+ E_i +K;
B_(i+1) ← A_i;
```

## INTEL® SHA EXTENSIONS

```
C_(i + 1) ← B_i ROL 30;  
D_(i + 1) ← C_i;  
E_(i + 1) ← D_i;  
ENDFOR
```

```
DEST[127:96] ← A_4;  
DEST[95:64] ← B_4;  
DEST[63:32] ← C_4;  
DEST[31:0] ← D_4;
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SHA1RND4: __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

| Opcode/<br>Instruction                      | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-------|------------------------------|--------------------------|---|
| 0F 38 C8 /r<br>SHA1NEXTE xmm1,<br>xmm2/m128 | RM    | V/V                          | SHA                      | Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1 |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 |
|-------|------------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        |

### Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

### Operation

#### SHA1NEXTE

$TMP \leftarrow (SRC1[127:96] \text{ ROL } 30);$

$DEST[127:96] \leftarrow SRC2[127:96] + TMP;$

$DEST[95:64] \leftarrow SRC2[95:64];$

$DEST[63:32] \leftarrow SRC2[63:32];$

$DEST[31:0] \leftarrow SRC2[31:0];$

### Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE: `__m128i __mm_sha1nexte_epu32(__m128i, __m128i);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

| Opcode/<br>Instruction                     | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| OF 38 C9 /r<br>SHA1MSG1 xmm1,<br>xmm2/m128 | RM    | V/V                          | SHA                      | Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1 |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 |
|-------|------------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        |

### Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

### Operation

#### SHA1MSG1

```

W0 ← SRC1[127:96];
W1 ← SRC1[95:64];
W2 ← SRC1[63:32];
W3 ← SRC1[31:0];
W4 ← SRC2[127:96];
W5 ← SRC2[95:64];

```

```

DEST[127:96] ← W2 XOR W0;
DEST[95:64] ← W3 XOR W1;
DEST[63:32] ← W4 XOR W2;
DEST[31:0] ← W5 XOR W3;

```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SHA1MSG1: __m128i_mm_sha1msg1_epu32(__m128i, __m128i);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.



## SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

| Opcode/<br>Instruction                     | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| OF 38 CA /r<br>SHA1MSG2 xmm1,<br>xmm2/m128 | RM    | V/V                          | SHA                      | Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1 |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 |
|-------|------------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        |

### Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

### Operation

#### SHA1MSG2

```

W13 ← SRC2[95:64];
W14 ← SRC2[63:32];
W15 ← SRC2[31:0];
W16 ← (SRC1[127:96] XOR W13) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63:32] XOR W15) ROL 1;
W19 ← (SRC1[31:0] XOR W16) ROL 1;

```

```

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;

```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA256RND2—Perform Two Rounds of SHA256 Operation

| Opcode/<br>Instruction                               | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| OF 38 CB /r<br>SHA256RND2 xmm1,<br>xmm2/m128, <XMM0> | RMO   | V/V                          | SHA                      | Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3         |
|-------|------------------|---------------|-------------------|
| RMI   | ModRM:reg (r, w) | ModRM:r/m (r) | Implicit XMM0 (r) |

### Description

The SHA256RND2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

### Operation

#### SHA256RND2

```
A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK0 ← XMM0[31:0];
WK1 ← XMM0[63:32];
```

FOR i = 0 to 1

```
A_(i+1) ← Ch(E_i, F_i, G_i) + Σ1( E_i ) + WKi + H_i + Maj(A_i, B_i, C_i) + Σ0( A_i);
B_(i+1) ← A_i;
C_(i+1) ← B_i;
D_(i+1) ← C_i;
E_(i+1) ← Ch(E_i, F_i, G_i) + Σ1( E_i ) + WKi + H_i + D_i;
F_(i+1) ← E_i;
G_(i+1) ← F_i;
H_(i+1) ← G_i;
```

ENDFOR

```
DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA256RNDSD: `__m128i_mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);`

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

| Opcode/<br>Instruction                       | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| OF 38 CC /r<br>SHA256MSG1 xmm1,<br>xmm2/m128 | RM    | V/V                          | SHA                      | Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1 |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 |
|-------|------------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        |

### Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

### Operation

#### SHA256MSG1

```
W4 ← SRC2[31: 0];
W3 ← SRC1[127:96];
W2 ← SRC1[95:64];
W1 ← SRC1[63: 32];
W0 ← SRC1[31: 0];
```

```
DEST[127:96] ← W3 +  $\sigma_0$ ( W4);
DEST[95:64] ← W2 +  $\sigma_0$ ( W3);
DEST[63:32] ← W1 +  $\sigma_0$ ( W2);
DEST[31:0] ← W0 +  $\sigma_0$ ( W1);
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SHA256MSG1: __m128i _mm_sha256msg1_epu32(__m128i, __m128i);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

| Opcode/<br>Instruction                       | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| 0F 38 CD /r<br>SHA256MSG2 xmm1,<br>xmm2/m128 | RM    | V/V                          | SHA                      | Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1 |

### Instruction Operand Encoding

| Op/En | Operand 1        | Operand 2     | Operand 3 |
|-------|------------------|---------------|-----------|
| RM    | ModRM:reg (r, w) | ModRM:r/m (r) | NA        |

### Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

### Operation

#### SHA256MSG2

$$\begin{aligned} W14 &\leftarrow \text{SRC2}[95:64]; \\ W15 &\leftarrow \text{SRC2}[127:96]; \\ W16 &\leftarrow \text{SRC1}[31:0] + \sigma_1(W14); \\ W17 &\leftarrow \text{SRC1}[63:32] + \sigma_1(W15); \\ W18 &\leftarrow \text{SRC1}[95:64] + \sigma_1(W16); \\ W19 &\leftarrow \text{SRC1}[127:96] + \sigma_1(W17); \end{aligned}$$

$$\begin{aligned} \text{DEST}[127:96] &\leftarrow W19; \\ \text{DEST}[95:64] &\leftarrow W18; \\ \text{DEST}[63:32] &\leftarrow W17; \\ \text{DEST}[31:0] &\leftarrow W16; \end{aligned}$$

### Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 : `_mm_sha256msg2_epu32(__m128i, __m128i);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

This page was  
intentionally left  
blank.

## CHAPTER 9

# ADDITIONAL NEW INSTRUCTIONS

---

This chapter describes additional new instructions for future Intel 64 processors.

### 9.1 INSTRUCTION FORMAT

The format used for describing each instruction as in the example below is described in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

### 9.2 INSTRUCTION SET REFERENCE

## PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

| Opcode/<br>Instruction     | Op/<br>En | 64/32 bit<br>Mode<br>Support | CPUID Feature<br>Flag | Description   |
|----------------------------|-----------|------------------------------|-----------------------|---|
| OF 0D /2<br>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.

This page was  
intentionally left  
blank.

# CHAPTER 10

## MEMORY INSTRUCTIONS

---

This chapter describes new instructions for future Intel processor generations that provide enhancements in application memory operation and the use of persistent memory.

### 10.1 DETECTION OF NEW INSTRUCTIONS

Hardware support for CLFLUSHOPT is indicated by:

- CPUID.(EAX=07H, ECX=0H):EBX.CLFLUSHOPT[bit 23]=1 indicates the processor supports CLFLUSHOPT instruction.

Hardware support for CLWB is indicated by:

- CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24]=1 indicates the processor supports CLWB instruction.

Software query the availability of PCOMMIT by examining:

- CPUID.(EAX=07H, ECX=0H):EBX.PCOMMIT[bit 22], If 1, indicates the processor supports PCOMMIT instruction.

### 10.2 PERSISTENT MEMORY

Persistent memory refers to Non-Volatile Memory (NVM) attached to the processor memory subsystem in a platform. Non-volatility may be achieved through the use of battery-backed volatile memory, or through non-volatile memory devices. The instruction set architecture enhancements described in this document is agnostic of the specific technology used to achieve non-volatility of memory. The enhancement is also agnostic to the specific usages of NVM, e.g. as fast paging device, block storage or cache emulation, or persistent memory. Support for specific persistent memory technologies is platform dependent.

#### 10.2.1 Accessing Persistent Memory

Persistent memory is accessible to software just like normal volatile main memory (DRAM), through the processor's load/store instructions. Accesses to persistent memory are subject to the same processor memory model (with respect to cacheability, coherency, processor memory ordering, memory types, etc.) as accesses to main volatile memory. This enables software to make use of the full breath of the IA instruction set architecture to program persistent memory.

Since existing software (that works with volatile main memory) has no concept of associating persistence to memory, persistent memory is reported distinctly from volatile main memory to system software (via ACPI, EFI interfaces). Depending on the system software architecture, persistent memory may be managed by the operating system memory manager, or may be managed through a block driver or file system driver in the storage software stack.

#### 10.2.2 Managing Persistence

Unlike volatile main memory, persistent memory may be used to store data durably, so that it is available across system failures and restarts. However, stores to persistent memory share the same volatile micro-architectural resources of the processor/platform as stores to volatile main memory. These include the processor store buffers, coherency caches (L1/L2/LLC etc.), any memory-side caches, on-chip and off-chip interconnect buffers, memory controller write buffers, etc. The data in a store to persistent memory becomes persistent (durable) only after it has either been written to the targeted non-volatile device, or to some intermediate power-fail protected storage/buffer. In case of a platform/power failure, the power-fail protection in the intermediate buffer guarantees that the residual energy/capacitance in the platform is utilized to drain written data to the backing non-volatile medium.

In order for software to make use of the persistence property of memory, software minimally needs the following support from the processor instruction set architecture:

1. **Failure atomicity for writes:** Failure atomicity refers to the maximum size (and alignment) of writes to persistent memory by software that is guaranteed to be atomic (all or nothing) in case of power or system failure. IA-32 and Intel-64 processors guarantee write atomicity for up to 64-bit accesses (aligned or unaligned) to cached memory that fit in a cache line (64-byte aligned region). For such writes, software can safely do in-place update of data in persistent memory and can assume all or nothing behavior. In-place updates of persistent memory helps improve performance, as software does not have to incur the overheads of copy-on-write or write-ahead-logging software techniques to guarantee write atomicity.
2. **Efficient cache flushing:** For performance reasons, accesses to persistent memory may be cached by the processor caches. While caches significantly improve memory access latencies and processor performance, caching requires software to ensure data from stores is indeed flushed from the volatile caches (written back to memory), as part of making it persistent. New optimized cache flush instructions that avoid the performance limitations of CLFLUSH are introduced. Section 10.4 provides the details of the CLFLUSHOPT and CLWB instructions.
  - CLFLUSHOPT is defined to provide efficient cache flushing.
  - CLWB instruction (Cache Line Write Back) writes back modified data of a cacheline similar to CLFLUSHOPT, but avoids invalidating the line from the cache (and instead transitions the line to non-modified state). CLWB attempts to minimize the compulsory cache miss if the same data is accessed temporally after the line is flushed.
3. **Committing to Persistence:** A store to persistent memory is not persistent until the store data reaches the non-volatile memory device or an intermediate power-fail protected buffer. While cache flushing ensures the data is out of the volatile caches, in modern platform architectures, the cache flush operation completes as soon as the modified data write back is posted to the memory subsystem write buffers (and before the data may have become persistent memory). Since the memory subsystem ensures the proper memory ordering rules are met (such as subsequent read of the written data is serviced from the write buffers), this posted behavior of writes is not visible to accesses to volatile memory. However this implies that to ensure writes to persistent memory are indeed committed to persistence, software needs to flush any volatile write buffers or caches in the memory subsystem. A new persistent commit instruction (PCOMMIT) is defined to commit write data queued in the memory subsystem to persistent memory. Section 10.4 provides the details of the PCOMMIT instruction.
4. **Non-temporal Store Optimization:** Software usages that require copying moderate to large amounts of data from volatile to persistent memory (or across persistent memory) may benefit from the weakly ordered non-temporal store operations (e.g., using MOVNTI instruction). Since the non-temporal store operations to write back mapped memory are guaranteed to always implicitly invalidate the line from the caches and issue the writes to memory (see Section 10.4.6.2. of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*), software can benefit from not having to explicitly flushing the caches after such writes and instead simply fence the weakly ordered writes followed by a persistent commit operation as described in Section 10.4.

Additional information on platform support of persistent memory configuration is described in Section 10.5.

## 10.3 INSTRUCTION FORMAT

The format used for describing each instruction as in the example below is described in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

**CLFLUSHOPT – Flush a Cache Line (THIS IS AN EXAMPLE)**

| Opcode/<br>Instruction       | Op/<br>En | 64/32bit<br>Mode<br>Support | CPUID<br>Feature Flag | Description                       |
|------------------------------|-----------|-----------------------------|-----------------------|-----------------------------------|
| 66 0F AE /7<br>CLFLUSHOPT m8 | M         | V/V                         | CLFLUSHOPT            | Flushes cache line containing m8. |

**Instruction Operand Encoding**

| Op/En | Operand 1     | Operand 2 | Operand 3 | Operand 4 |
|-------|---------------|-----------|-----------|-----------|
| M     | ModRM:r/m (r) | NA        | NA        | NA        |

**10.4 INSTRUCTION SET REFERENCE**

## CLFLUSHOPT—Flush a Cache Line Optimized

| Opcode/<br>Instruction       | Op/<br>En | 64/32 bit<br>Mode<br>Support | CPUID<br>Feature Flag | Description                       |
|------------------------------|-----------|------------------------------|-----------------------|-----------------------------------|
| 66 OF AE /7<br>CLFLUSHOPT m8 | M         | V/V                          | CLFLUSHOPT            | Flushes cache line containing m8. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3 | Operand 4 |
|-------|---------------|-----------|-----------|-----------|
| M     | ModRM:r/m (w) | NA        | NA        | NA        |

### Description

Writes back to memory the cache line (if dirty) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line is always invalidated from the processor cache hierarchy. The source operand is a byte memory location.

The availability of CLFLUSHOPT instruction is indicated by the presence of the CPUID feature flag CLFLUSHOPT (bit 23 of the EBX register, see “CPUID — CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHH instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSHOPT instruction is not ordered with respect to PREFETCHH instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSHOPT instruction that references the cache line).

CLFLUSHOPT instruction is ordered only by store-fencing operations. For example, software can use an SFENCE, MFENCE, XCHG, or LOCK-prefixed instructions to ensure that previous stores are included in the write-back. CLFLUSHOPT instruction need not be ordered by another CLFLUSHOPT or CLWB instruction. CLFLUSHOPT is implicitly ordered with older stores executed by the logical processor to the same cacheline address.

For usages that require efficient flushing of multiple cache lines, software is recommended to use CLFLUSHOPT (with appropriate fencing) instead of CLFLUSH for improved performance.

Executions of CLFLUSHOPT interact with executions of PCOMMIT. The PCOMMIT instruction operates on certain store-to-memory operations that have been accepted to memory (see description of “PCOMMIT—Persistent Commit” for additional information on accepted to memory). CLFLUSHOPT executed for the same cache line as an older store causes the store to become accepted to memory when the CLFLUSHOPT execution becomes globally visible.

The CLFLUSHOPT instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLFLUSHOPT instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSHOPT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLFLUSHOPT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSHOPT instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

### Operation

Flush\_Cache\_Line\_Optimized(m8);

### Flags Affected

None.

**C/C++ Compiler Intrinsic Equivalent**

```
CLFLUSHOPT void _mm_clflushopt(void const *p);
```

**Protected Mode Exceptions**

|                 |   |
|-----------------|---|
| #UD             | If the LOCK prefix is used.<br>If CPUID.(EAX=07H, ECX=0H):EBX.CLFLUSHOPT[bit 23] = 0. |
| #GP(0)          | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0)          | For an illegal address in the SS segment.   |
| #PF(fault-code) | For a page fault.   |

**Real-Address Mode Exceptions**

|     |   |
|-----|---|
| #UD | If the LOCK prefix is used.<br>If CPUID.(EAX=07H, ECX=0H):EBX.CLFLUSHOPT[bit 23] = 0. |
| #GP | If any part of the operand lies outside the effective address space from 0 to FFFFH.  |

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

|                 |                   |
|-----------------|-------------------|
| #PF(fault-code) | For a page fault. |
|-----------------|-------------------|

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

|                 |   |
|-----------------|---|
| #UD             | If the LOCK prefix is used.<br>If CPUID.(EAX=07H, ECX=0H):EBX.CLFLUSHOPT[bit 23] = 0. |
| #SS(0)          | If a memory address referencing the SS segment is in a non-canonical form.            |
| #GP(0)          | If the memory address is in a non-canonical form.                                     |
| #PF(fault-code) | For a page fault.   |

## CLWB—Cache Line Write Back

| Opcode/<br>Instruction | Op/<br>En | 64/32 bit<br>Mode<br>Support | CPUID<br>Feature Flag | Description  |
|------------------------|-----------|------------------------------|-----------------------|--|
| 66 0F AE /6<br>CLWB m8 | M         | V/V                          | CLWB                  | Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3 | Operand 4 |
|-------|---------------|-----------|-----------|-----------|
| M     | ModRM:r/m (w) | NA        | NA        | NA        |

### Description

Writes back to memory the cache line (if dirty) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

The availability of CLWB instruction is indicated by the presence of the CPUID feature flag CLWB (bit 24 of the EBX register, see “CPUID — CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHH instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLWB instruction is not ordered with respect to PREFETCHH instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLWB instruction that references the cache line).

CLWB instruction is ordered only by store-fencing operations. For example, software can use an SFENCE, MFENCE, XCHG, or LOCK-prefixed instructions to ensure that previous stores are included in the write-back. CLWB instruction need not be ordered by another CLWB or CLFLUSHOPT instruction. CLWB is implicitly ordered with older stores executed by the logical processor to the same address.

For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

Executions of CLWB interact with executions of PCOMMIT. The PCOMMIT instruction operates on certain store-to-memory operations that have been accepted to memory. CLWB executed for the same cache line as an older store causes the store to become accepted to memory when the CLWB execution becomes globally visible.

The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

### Operation

Cache\_Line\_Write\_Back(m8);

### Flags Affected

None.



**C/C++ Compiler Intrinsic Equivalent**

```
CLWB void _mm_clwb(void const *p);
```

**Protected Mode Exceptions**

|                 |   |
|-----------------|---|
| #UD             | If the LOCK prefix is used.<br>If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.       |
| #GP(0)          | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0)          | For an illegal address in the SS segment.   |
| #PF(fault-code) | For a page fault.   |

**Real-Address Mode Exceptions**

|     |  |
|-----|--|
| #UD | If the LOCK prefix is used.<br>If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.      |
| #GP | If any part of the operand lies outside the effective address space from 0 to FFFFH. |

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

|                 |                   |
|-----------------|-------------------|
| #PF(fault-code) | For a page fault. |
|-----------------|-------------------|

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

|                 |   |
|-----------------|---|
| #UD             | If the LOCK prefix is used.<br>If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0. |
| #SS(0)          | If a memory address referencing the SS segment is in a non-canonical form.      |
| #GP(0)          | If the memory address is in a non-canonical form.                               |
| #PF(fault-code) | For a page fault.   |

## PCOMMIT—Persistent Commit

| Opcode/<br>Instruction | Op/<br>En | 64/32 bit<br>Mode<br>Support | CPUID<br>Feature Flag | Description                          |
|------------------------|-----------|------------------------------|-----------------------|--------------------------------------|
| 66 0F AE F8<br>PCOMMIT | NP        | V/V                          | PCOMMIT               | Commits stores to persistent memory. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|-------|-----------|-----------|-----------|-----------|
| NP    | NA        | NA        | NA        | NA        |

### Description

The PCOMMIT instruction causes certain store-to-memory operations to persistent memory ranges to become persistent (power failure protected).<sup>1</sup> Specifically, PCOMMIT applies to those stores that have been accepted to memory.

While all store-to-memory operations are eventually accepted to memory, the following items specify the actions software can take to ensure that they are accepted:

- Non-temporal stores to write-back (WB) memory and all stores to uncacheable (UC), write-combining (WC), and write-through (WT) memory are accepted to memory as soon as they are globally visible.
- If, after an ordinary store to write-back (WB) memory becomes globally visible, CLFLUSH, CLFLUSHOPT, or CLWB is executed for the same cache line as the store, the store is accepted to memory when the CLFLUSH, CLFLUSHOPT or CLWB execution itself becomes globally visible.

If PCOMMIT is executed after a store to a persistent memory range is accepted to memory, the store becomes persistent when the PCOMMIT becomes globally visible. This implies that, if an execution of PCOMMIT is globally visible when a later store to persistent memory is executed, that store cannot become persistent before the stores to which the PCOMMIT applies.

The following items detail the ordering between PCOMMIT and other operations:

- A logical processor does **not** ensure previous stores and executions of CLFLUSHOPT and CLWB (by that logical processor) are globally visible before commencing an execution of PCOMMIT. This implies that software must use appropriate fencing instruction (e.g., SFENCE) to ensure the previous stores-to-memory operations and CLFLUSHOPT and CLWB executions to persistent memory ranges are globally visible (so that they are accepted to memory), before executing PCOMMIT.
- A logical processor does **not** ensure that an execution of PCOMMIT is globally visible before commencing subsequent stores. Software that requires that such stores not become globally visible before PCOMMIT (e.g., because the younger stores must not become persistent before those committed by PCOMMIT) can ensure by using an appropriate fencing instruction (e.g., SFENCE) between PCOMMIT and the later stores.
- An execution of PCOMMIT is ordered with respect to executions of SFENCE, MFENCE, XCHG or LOCK-prefixed instructions, and serializing instructions (e.g., CPUID).
- Executions of PCOMMIT are not ordered with respect to load operations. Software can use MFENCE to order loads with PCOMMIT.
- Executions of PCOMMIT do not serialize the instruction stream.

The PCOMMIT instruction can be used at all privilege levels and the instruction's operation is the same in non-64-bit modes and 64-bit mode.

1. A platform may support one or more persistent memory ranges and report those ranges to system software. The power-fail protection or persistence may be implemented through platform-specific means such as use of battery-backed volatile memory, use of non-volatile memory, etc.

In some implementations, the PCOMMIT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). PCOMMIT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on PCOMMIT instruction to force a transactional abort, since whether they cause transactional aborts is implementation dependent.

### Operation

Commit\_To\_Persistence;

### Flags Affected

None.

### C/C++ Compiler Intrinsic Equivalent

PCOMMIT void \_mm\_pcommit(void );

### Exceptions (All Operating Modes)

#UD                    If the LOCK prefix is used.  
                         If CPUID.(EAX=07H, ECX=0H):EBX.PCOMMIT[bit 22] = 0.

## 10.5 PERSISTENT MEMORY CONFIGURATION AND ENUMERATION OF PLATFORM SUPPORT

Persistent memory configuration in a platform requires BIOS support to configure the memory controllers, memory devices (DIMMs), system address decoders etc., and to report persistent memory distinct from volatile main memory to system software. PCOMMIT execution is relevant on a platform only when the platform is configured with persistent memory.

In addition to requirement of BIOS to manage memory controllers, memory devices, and system address decoders, BIOS sets IA32\_FEATURE\_CONTROL.PCOMMIT\_ENABLE[bit 19] and IA32\_FEATURE\_CONTROL.LOCK to indicate platform support of processor enumeration of PCOMMIT instruction using CPUID. Since the IA32\_FEATURE\_CONTROL MSR contains multiple feature enable bits, BIOS must lock the MSR only after all appropriate feature enable bits in IA32\_FEATURE\_CONTROL MSR are programmed.

On processors that supports PCOMMIT, PCOMMIT is enumerated through CPUID (CPUID.7.0.EBX[22]) only when the feature is enabled by BIOS. i.e, CPUID.(EAX=7, ECX=0):EBX[22] reflects the state of

IA32\_FEATURE\_CONTROL[19] & IA32\_FEATURE\_CONTROL[0].

PCOMMIT execution will always #UD if CPUID.07H:EBX.PCOMMIT[bit 22] = 0.

## 10.6 PCOMMIT – VIRTUALIZATION SUPPORT

Persistent memory may be virtualized for software running inside a Virtual Machine (VM) by a Virtual Machine Monitor (VMM). Depending on usage models, the virtualization of persistent memory can take different forms, such as VMM simply partitioning up the persistent memory across one or more VMs, or emulating or over-committing persistent memory for a VM, or replicating persistent memory modifications to a remote node. Some of these usages may require direct execution of PCOMMIT by guest software running inside a VM, while other usages may require or benefit from VMM intercepting guest execution of PCOMMIT instruction.

To support such usages, a new secondary processor-based VM-execution control called “PCOMMIT exiting” (bit 21) is defined.

IA32\_VMX\_PROCBASED\_CTL2[53] (which enumerates support for the 1-setting of “PCOMMIT exiting”) is always the same as CPUID.07H:EBX.PCOMMIT[bit 22]. Thus, software can set “PCOMMIT exiting” to 1 if and only if the PCOMMIT instruction is enumerated via CPUID (which requires IA32\_FEATURE\_CONTROL[19] and IA32\_FEATURE\_CONTROL[0] to be both 1).

When IA32\_VMX\_PROCBASED\_CTL2[53] = 0, VM entry will fail early if “PCOMMIT exiting” is 1.

When ‘PCOMMIT exiting’ execution control is set, PCOMMIT execution will cause VM exit with a new basic exit reason (41H) called “PCOMMIT”.

## 10.7 PCOMMIT AND SGX INTERACTION

PCOMMIT execution within an enclave always #UD. This is regardless of CPUID enumeration and VM-execution control for PCOMMIT. The #UD happens with priority above any VM exit on PCOMMIT instruction execution.

# INDEX

## A

- ADDPD - Add Packed Double-Precision Floating-Point Values 5-10
- ADDPS- Add Packed Single-Precision Floating-Point Values 5-13
- ADDSD- Add Scalar Double-Precision Floating-Point Values 5-16
- ADDSS- Add Scalar Single-Precision Floating-Point Values 5-18
- ANDNPD- Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values 5-35
- ANDNPS- Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values 5-38
- ANDPD- Bitwise Logical AND of Packed Double Precision Floating-Point Values 5-29
- ANDPS- Bitwise Logical AND of Packed Single Precision Floating-Point Values 5-32

## B

- Brand information 2-32
  - processor brand index 2-34
  - processor brand string 2-32

## C

- Cache and TLB information 2-28
- Cache Inclusiveness 2-14
- CLFLUSH instruction
  - CPUID flag 2-27
- CMOVcc flag 2-27
- CMOVcc instructions
  - CPUID flag 2-27
- CMPPD- Compare Packed Double-Precision Floating-Point Values 5-60
- CMPPS- Compare Packed Single-Precision Floating-Point Values 5-67
- CMPSD- Compare Scalar Double-Precision Floating-Point Values 5-73
- CMPSS- Compare Scalar Single-Precision Floating-Point Values 5-77
- CMPXCHG16B instruction
  - CPUID bit 2-25
- CMPXCHG8B instruction
  - CPUID flag 2-27
- COMISD- Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS 5-82
- COMISS- Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS 5-84
- CPUID instruction 2-12, 2-27
  - 36-bit page size extension 2-27
  - APIC on-chip 2-27
  - basic CPUID information 2-13
  - cache and TLB characteristics 2-13, 2-28
  - CLFLUSH flag 2-27
  - CLFLUSH instruction cache line size 2-23
  - CMPXCHG16B flag 2-25
  - CMPXCHG8B flag 2-27
  - CPL qualified debug store 2-24
  - debug extensions, CR4.DE 2-26
  - debug store supported 2-27
  - deterministic cache parameters leaf 2-13, 2-16, 2-17, 2-18
  - extended function information 2-19
  - feature information 2-26
  - FPU on-chip 2-26
  - FSAVE flag 2-27
  - FXRSTOR flag 2-27
  - HT technology flag 2-28
  - IA-32e mode available 2-20
  - input limits for EAX 2-21
  - L1 Context ID 2-25
  - local APIC physical ID 2-23
  - machine check architecture 2-27
  - machine check exception 2-27
  - memory type range registers 2-27
  - MONITOR feature information 2-31
  - MONITOR/MWAIT flag 2-24

MONITOR/MWAIT leaf 2-14, 2-15, 2-16, 2-17, 2-19  
 MWAIT feature information 2-31  
 page attribute table 2-27  
 page size extension 2-26  
 performance monitoring features 2-31  
 physical address bits 2-20  
 physical address extension 2-27  
 power management 2-31, 2-32  
 processor brand index 2-23, 2-32  
 processor brand string 2-20, 2-32  
 processor serial number 2-27  
 processor type field 2-22  
 PTE global bit 2-27  
 RDMSR flag 2-26  
 returned in EBX 2-23  
 returned in ECX & EDX 2-23  
 self snoop 2-28  
 SpeedStep technology 2-24  
 SS2 extensions flag 2-28  
 SSE extensions flag 2-28  
 SSE3 extensions flag 2-24  
 SSSE3 extensions flag 2-24  
 SYSENTER flag 2-27  
 SYSEXIT flag 2-27  
 thermal management 2-31, 2-32  
 thermal monitor 2-24, 2-27, 2-28  
 time stamp counter 2-26  
 using CPUID 2-12  
 vendor ID string 2-21  
 version information 2-13, 2-30  
 virtual 8086 Mode flag 2-26  
 virtual address bits 2-20  
 WRMSR flag 2-26  
 CVTDQ2PD- Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values 5-100, 5-114, 5-118, 5-139, 5-141, 5-162, 5-166, 5-188, 5-190  
 CVTDQ2PS- Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values 5-103  
 CVTPD2DQ- Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 5-106  
 CVTPD2PS- Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values 5-110  
 CVTPS2DQ- Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values 5-127  
 CVTPS2DQ- Convert Packed Single Precision Floating-Point Values to Packed Singed Doubleword Integer Values 5-132, 5-134, 5-173, 5-175  
 CVTPS2PD- Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values 5-136  
 CVTSD2SI- Convert Scalar Double Precision Floating-Point Value to Doubleword Integer 5-143  
 CVTSD2SS- Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value 5-147  
 CVTSI2SD- Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value 5-114, 5-118, 5-139, 5-141, 5-149, 5-162, 5-166, 5-188, 5-190  
 CVTSI2SS- Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value 5-151  
 CVTSS2SD- Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value 5-153  
 CVTSS2SI- Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer 5-155  
 CVTTPD2DQ- Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers 5-159  
 CVTTPS2DQ- Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values 5-168  
 CVTTS2SI- Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer 5-177  
 CVTSS2SI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer 5-180

## D

DIVPD- Divide Packed Double-Precision Floating-Point Values 5-25, 5-86  
 DIVPS- Divide Packed Single-Precision Floating-Point Values 5-89  
 DIVSD- Divide Scalar Double-Precision Floating-Point Values 5-92  
 DIVSS- Divide Scalar Single-Precision Floating-Point Values 5-94

## E

EVEX.R 5-4  
 EXTRACTPS- Extract packed floating-point values 5-215

## F

Feature information, processor 2-12

FXRSTOR instruction

    CPUID flag 2-27

FXSAVE instruction

    CPUID flag 2-27

## H

Hyper-Threading Technology

    CPUID flag 2-28

## I

IA-32e mode

    CPUID flag 2-20

INSERTPS- Insert Scalar Single-Precision Floating-Point Value 5-383

## L

L1 Context ID 2-25

## M

Machine check architecture

    CPUID flag 2-27

    description 2-27

MAXPD- Maximum of Packed Double-Precision Floating-Point Values 5-386

MAXPS- Maximum of Packed Single-Precision Floating-Point Values 5-389

MAXSD- Return Maximum Scalar Double-Precision Floating-Point Value 1-4, 5-392

MAXSS- Return Maximum Scalar Single-Precision Floating-Point Value 5-394

MINPD- Minimum of Packed Double-Precision Floating-Point Values 5-396

MINPS- Minimum of Packed Single-Precision Floating-Point Values 5-399

MINSD- Return Minimum Scalar Double-Precision Floating-Point Value 5-402

MINSS- Return Minimum Scalar Single-Precision Floating-Point Value 5-404

MMX instructions

    CPUID flag for technology 2-27

Model & family information 2-30

MONITOR instruction

    CPUID flag 2-24

    feature data 2-31

MOVAPD- Move Aligned Packed Double-Precision Floating-Point Values 5-406

MOVAPS- Move Aligned Packed Single-Precision Floating-Point Values 5-410

MOVD/MOVQ- Move Doubleword and Quadword 5-414

MOVDDUP- Replicate Double FP Values 5-420

MOVDQA- Move Aligned Packed Integer Values 5-423

MOVDQU- Move Unaligned Packed Integer Values 5-428

MOVHLPS - Move Packed Single-Precision Floating-Point Values High to Low 5-435

MOVHPD- Move High Packed Double-Precision Floating-Point Values 5-437

MOVHPS- Move High Packed Single-Precision Floating-Point Values 5-439

MOVLPD- Move Low Packed Double-Precision Floating-Point Values 5-443

MOVLPS- Move Low Packed Single-Precision Floating-Point Values 5-445

MOVNTDQ- Store Packed Integers Using Non-Temporal Hint 5-449

MOVNTPD- Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint 5-451

MOVNTPS- Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint 5-453

MOVQ- Move Quadword 5-417

MOVSD- Move or Merge Scalar Double-Precision Floating-Point Value 5-455

MOVSHDUP- Replicate Single FP Values 5-458

MOVSLDUP- Replicate Single FP Values 5-461

MOVSS- Move or Merge Scalar Single-Precision Floating-Point Value 5-464

MOVUPD- Move Unaligned Packed Double-Precision Floating-Point Values 5-467

MOVUPS- Move Unaligned Packed Single-Precision Floating-Point Values 5-471

MULPD- Multiply Packed Double-Precision Floating-Point Values 5-478

MULPS- Multiply Packed Single-Precision Floating-Point Values 5-481

MULSD- Multiply Scalar Double-Precision Floating-Point Values 5-484

MULSS- Multiply Scalar Single-Precision Floating-Point Values 5-486

MWAIT instruction

CPUID flag 2-24  
feature data 2-31

## O

ORPD- Bitwise Logical OR of Packed Double Precision Floating-Point Values 5-488  
ORPS- Bitwise Logical OR of Packed Single Precision Floating-Point Values 5-491

## P

PASB instruction 5-196, 5-500, 5-508, 5-513, 5-525, 5-529, 5-542, 5-675, 5-678, 5-701, 5-766  
PASB/PASW/PASD/PASQ - Packed Absolute Value 5-494  
PABSD instruction 5-196, 5-500, 5-508, 5-513, 5-525, 5-529, 5-542, 5-675, 5-678, 5-701, 5-766  
PABS instruction 5-196, 5-500, 5-508, 5-513, 5-525, 5-529, 5-542, 5-675, 5-678, 5-701, 5-766  
PADDB/PADDW/PADDD/PADDQ - Add Packed Integers 5-518  
PAND - Logical AND 5-536  
PANDN - Logical AND NOT 5-539  
PCLMULQDQ - Carry-Less Multiplication Quadword 5-562, 5-571  
Pending break enable 2-28  
Performance-monitoring counters  
    CPUID inquiry for 2-31  
PEXTRB/PEXTRW/PEXTRD/PEXTRQ- Extract Integer 5-625  
PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint 9-2  
PTEST- Packed Bit Test 6-17

## R

RDMSR instruction  
    CPUID flag 2-26  
ROUNDPD- Round Packed Double-Precision Floating-Point Values 5-958

## S

Self Snoop 2-28  
SHUFF32x4/SHUFF64x2/SHUFI32x4/SHUFI64x2 - Shuffle Packed Values at 128-bit Granularity 5-859  
SHUFPD - Shuffle Packed Double Precision Floating-Point Values 5-864, 5-958  
SHUFPS - Shuffle Packed Single Precision Floating-Point Values 5-869  
SpeedStep technology 2-24  
SQRTPD- Square Root of Double-Precision Floating-Point Values 5-958  
SQRTPD—Square Root of Double-Precision Floating-Point Values 5-873  
SQRTPS- Square Root of Single-Precision Floating-Point Values 5-876  
SQRTSD - Compute Square Root of Scalar Double-Precision Floating-Point Value 5-879, 5-958  
SQRTSS - Compute Square Root of Scalar Single-Precision Floating-Point Value 5-881  
SSE extensions  
    CPUID flag 2-28  
SSE2 extensions  
    CPUID flag 2-28  
SSE3  
    CPUID flag 2-24  
SSE3 extensions  
    CPUID flag 2-24  
SSSE3 extensions  
    CPUID flag 2-24  
Stepping information 2-30  
SUBPD- Subtract Packed Double Precision Floating-Point Values 5-958  
SUBPD- Subtract Packed Double-Precision Floating-Point Values 5-958  
SUBPS- Subtract Packed Single-Precision Floating-Point Values 5-961  
SUBSD- Subtract Scalar Double-Precision Floating-Point Values 5-964  
SUBSS- Subtract Scalar Single-Precision Floating-Point Values 5-966  
SYSENTER instruction  
    CPUID flag 2-27  
SYSEXIT instruction  
    CPUID flag 2-27

## T

Thermal Monitor  
    CPUID flag 2-28



Thermal Monitor 2 2-24  
CPUID flag 2-24  
Time Stamp Counter 2-26

## U

UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS 5-968  
UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS 5-970  
UNPCKHPD- Unpack and Interleave High Packed Double-Precision Floating-Point Values 5-972  
UNPCKHPS- Unpack and Interleave High Packed Single-Precision Floating-Point Values 5-976  
UNPCKLPD- Unpack and Interleave Low Packed Double-Precision Floating-Point Values 5-980  
UNPCKLPS- Unpack and Interleave Low Packed Single-Precision Floating-Point Values 5-984

## V

VALIGND/VALIGNQ- Align Doubleword/Quadword Vectors 5-20  
VBLENDMPD- Blend Float64 Vectors Using an OpMask Control 5-23  
VCOMPRESSPD- Store Sparse Double-Precision Floating-Point Values into Dense Memory 5-96  
VCOMPRESSPD- Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory 5-96  
VCVTPD2UDQ- Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers 5-116  
VCVTPS2UDQ- Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values 5-130  
VCVTSD2USI- Convert Scalar Double Precision Floating-Point Value to Unsigned Doubleword Integer 5-145  
VCVTSS2USI- Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer 5-157  
VCVTTPD2UDQ- Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers 5-164  
VCVTTPS2UDQ- Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values 5-171  
VCVTSD2USI- Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer 5-179  
VCVTSS2USI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer 5-182  
VCVTUDQ2PD- Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values 5-184  
VCVTUDQ2PS- Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values 5-186  
VCVTUSI2SD- Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value 5-192  
VCVTUSI2SS- Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value 5-194  
Version information, processor 2-12  
VEX 5-2  
VEX.B 5-2  
VEX.L 5-2, 5-4  
VEX.mmmmm 5-2  
VEX.pp 5-3, 5-4  
VEX.R 5-4  
VEX.vvvv 5-2  
VEX.W 5-2  
VEX.X 5-2  
VEXP2PD—Approximation to the Exponential  $2^x$  of Packed Double-Precision Floating-Point Values with Less Than  $2^{-23}$  Relative Error 7-3  
VEXP2PS—Approximation to the Exponential  $2^x$  of Packed Single-Precision Floating-Point Values with Less Than  $2^{-23}$  Relative Error 7-5  
VEXTRACTF128- Extract Packed Floating-Point Values 5-203  
VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 5-246  
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values 5-249  
VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values 5-256  
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values 5-283  
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values 5-289  
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS - Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values 5-292  
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values 5-263  
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 5-295  
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values 5-302  
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values 5-308  
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values 5-326

VGATHERDPS/VGATHERDPD - Gather Packed Single, Packed Double with Signed Dword 5-347  
 VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD - Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint 7-23  
 VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD - Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint 7-25  
 VGATHERQPS/VGATHERQPD - Gather Packed Single, Packed Double with Signed Qword Indices 5-350  
 VINSERTF128/VINSERTF32x4/VINSERTF64x4- Insert Packed Floating-Point Values 5-375  
 VINSERTI128/VINSERTI32x4/VINSERTI64x4- Insert Packed Integer Values 5-379  
 VPBLENDMD- Blend Int32 Vectors Using an OpMask Control 5-27  
 VPBROADCASTM—Broadcast Mask to Vector Register 5-546  
 VPCMPD/VPCMPUD - Compare Packed Integer Values into Mask 5-565  
 VPCMPQ/VPCMPUQ - Compare Packed Integer Values into Mask 5-568  
 VPCONFLICTD/Q - Detect Conflicts Within a Vector of Packed Dword, Packed Qword Values into Dense Memory/Register 7-3  
 VPERMILPD- Permute Double-Precision Floating-Point Values 5-602  
 VPERMILPS- Permute Single-Precision Floating-Point Values 5-607  
 VPEXPANDD- Load Sparse Packed Doubleword Integer Values from Dense Memory/Register 5-594  
 VPGATHERDD/VPGATHERDQ- Gather Packed Dword, Packed Qword with Signed Dword Indices 5-341  
 VPGATHERQD/VPGATHERQQ- Gather Packed Dword, Packed Qword with Signed Qword Indices 5-344  
 VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values 5-628  
 VPMOVDDB/VPMOVSDDB/VPMOVUSDB - Down Convert DWord to Byte 5-693  
 VPMOVDW/VPMOVSDW/VPMOVUSDW - Down Convert DWord to Word 5-697  
 VPMOVQB/VPMOVSQB/VPMOVUSQB - Down Convert QWord to Byte 5-681  
 VPMOVQD/VPMOVSQD/VPMOVUSQD - Down Convert QWord to DWord 5-689  
 VPMOVQW/VPMOVSQW/VPMOVUSQW - Down Convert QWord to Word 5-685  
 VPTERNLOGD/VPTERNLOGQ - Bitwise Ternary Logic 5-883  
 VPTESTMD/VPTESTMQ - Logical AND and Set Mask 5-886  
 VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than 2<sup>-28</sup> Relative Error 7-7  
 VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than 2<sup>-28</sup> Relative Error 7-11  
 VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2<sup>-28</sup> Relative Error 7-9  
 VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than 2<sup>-28</sup> Relative Error 7-13  
 VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than 2<sup>-28</sup> Relative Error 7-15  
 VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than 2<sup>-28</sup> Relative Error 7-19  
 VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than 2<sup>-28</sup> Relative Error 7-17  
 VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2<sup>-28</sup> Relative Error 7-21  
 VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write 7-27  
 VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write 7-29

## W

WBINVD/INVD bit 2-14  
 WRMSR instruction  
 CPUID flag 2-26

## X

XFEATURE\_ENALBED\_MASK 2-1  
 XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values 5-988  
 XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values 5-991  
 XRSTOR 1-1, 2-1, 2-31, 5-7  
 XSAVE 1-1, 2-1, 2-4, 2-25, 2-31, 5-7