

Intel® Advanced Vector Extensions Programming Reference

AUGUST 2008

319433-003

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

Intel may make changes to specifications and product descriptions at any time, without notice.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

The Intel® 64 architecture processors may contain design defects or errors known as errata. Current characterized errata are available on request.

Hyper-Threading Technology requires a computer system with an Intel® processor supporting Hyper-Threading Technology and an HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information, see <http://www.intel.com/technology/hyperthread/index.htm>; including details on which processors support HT Technology.

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and for some uses, certain platform software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations. Intel® Virtualization Technology-enabled BIOS and VMM applications are currently in development.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Processors will not operate (including 32-bit operation) without an Intel® 64 architecture-enabled BIOS. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Intel, Pentium, Intel Xeon, Intel NetBurst, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo, Intel Core 2 Extreme, Intel Pentium D, Itanium, Intel SpeedStep, MMX, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 5937
Denver, CO 80217-9808

or call 1-800-548-4725
or visit Intel's website at <http://www.intel.com>

Copyright © 1997-2008 Intel Corporation

TABLE OF CONTENT

CHAPTER 1 INTEL® ADVANCED VECTOR EXTENSIONS

| | | |
|-------|--|-----|
| 1.1 | About This Document | 1-1 |
| 1.2 | Overview | 1-1 |
| 1.3 | Intel® Advanced Vector Extensions Architecture Overview | 1-2 |
| 1.3.1 | 256-Bit Wide SIMD Register Support | 1-2 |
| 1.3.2 | Instruction Syntax Enhancements | 1-3 |
| 1.3.3 | VEX Prefix Instruction Encoding Support | 1-4 |
| 1.4 | Functional Overview | 1-4 |
| 1.4.1 | 256-bit Floating-Point Arithmetic Processing Enhancements | 1-5 |
| 1.4.2 | 256-bit Non-Arithmetic Instruction Enhancements | 1-5 |
| 1.4.3 | Arithmetic Primitives for 128-bit Vector and Scalar processing | 1-6 |
| 1.4.4 | Non-Arithmetic Primitives for 128-bit Vector and Scalar Processing | 1-6 |
| 1.5 | General Encryption and Cryptographic Processing | 1-7 |

CHAPTER 2 APPLICATION PROGRAMMING MODEL

| | | |
|-------|--|------|
| 2.1 | Detection of PCLMULQDQ and AES Instructions | 2-9 |
| 2.2 | Detection of AVX and FMA Instructions | 2-9 |
| 2.2.1 | Detection of FMA | 2-11 |
| 2.2.2 | Detection of VEX-Encoded AES | 2-12 |
| 2.3 | Fused-Multiply-ADD (FMA) Numeric Behavior | 2-12 |
| 2.4 | Accessing YMM Registers | 2-17 |
| 2.5 | Memory alignment | 2-17 |
| 2.6 | SIMD floating-point Exceptions | 2-20 |
| 2.7 | Instruction Exception Specification | 2-21 |
| 2.7.1 | Exceptions Type 1 (Aligned memory reference) | 2-26 |
| 2.7.2 | Exceptions Type 2 (≥ 16 Byte Memory Reference, Unaligned) | 2-26 |
| 2.7.3 | Exceptions Type 3 (< 16 Byte memory argument) | 2-27 |
| 2.7.4 | Exceptions Type 4 (≥ 16 Byte mem arg no alignment, no floating-point exceptions) | 2-28 |
| 2.7.5 | Exceptions Type 5 (< 16 Byte mem arg and no FP exceptions) | 2-29 |
| 2.7.6 | Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues) | 2-30 |
| 2.7.7 | Exceptions Type 7 (No FP exceptions, no memory arg) | 2-31 |
| 2.7.8 | Exceptions Type 8 (AVX and no memory argument) | 2-32 |
| 2.7.9 | Exception Type 9 (AVX) | 2-33 |
| 2.8 | Programming Considerations with 128-bit SIMD Instructions | 2-33 |
| 2.8.1 | Clearing Upper YMM State Between AVX and Legacy SSE Instructions | 2-34 |
| 2.8.2 | Using AVX 128-bit Instructions Instead of Legacy SSE instructions | 2-35 |
| 2.8.3 | Unaligned Memory Access and Buffer Size Management | 2-35 |
| 2.9 | CPUID Instruction | 2-35 |
| | CPUID—CPU Identification | 2-36 |

CHAPTER 3 SYSTEM PROGRAMMING MODEL

| | | |
|-------|---|------|
| 3.1 | YMM State, VEX Prefix and Supported Operating Modes | 3-69 |
| 3.2 | YMM State Management | 3-70 |
| 3.2.1 | Detection of YMM State Support..... | 3-70 |
| 3.2.2 | Enabling of YMM State | 3-70 |
| 3.2.3 | Enabling of SIMD Floating-Exception Support | 3-71 |
| 3.2.4 | The Layout of XSAVE Area | 3-72 |
| 3.2.5 | XSAVE/XRESTOR Interaction with YMM State and MXCSR..... | 3-73 |
| 3.3 | Reset Behavior | 3-74 |
| 3.4 | Emulation..... | 3-75 |
| 3.5 | Writing AVX floating-point exception handlers..... | 3-75 |

CHAPTER 4 INSTRUCTION FORMAT

| | | |
|---------|--|------|
| 4.1 | Instruction Formats | 4-77 |
| 4.1.1 | VEX and the LOCK prefix | 4-78 |
| 4.1.2 | VEX and the 66H, F2H, and F3H prefixes..... | 4-78 |
| 4.1.3 | VEX and the REX prefix | 4-78 |
| 4.1.4 | The VEX Prefix | 4-78 |
| 4.1.4.1 | VEX Byte 0, bits[7:0]..... | 4-82 |
| 4.1.4.2 | VEX Byte 1, bit [7] - 'R' | 4-82 |
| 4.1.4.3 | 3-byte VEX byte 1, bit[6] - 'X'..... | 4-82 |
| 4.1.4.4 | 3-byte VEX byte 1, bit[5] - 'B'..... | 4-82 |
| 4.1.4.5 | 3-byte VEX byte 2, bit[7] - 'W' | 4-82 |
| 4.1.4.6 | 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or dest Register Specifier | 4-83 |
| 4.1.5 | Instruction Operand Encoding and VEX.vvvv, ModR/M..... | 4-84 |
| 4.1.5.1 | 3-byte VEX byte 1, bits[4:0] - "m- <i>mmmm</i> " | 4-87 |
| 4.1.5.2 | 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- " <i>L</i> " | 4-87 |
| 4.1.5.3 | 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- " <i>pp</i> " | 4-88 |
| 4.1.6 | The Opcode Byte | 4-88 |
| 4.1.7 | The MODRM, SIB, and Displacement Bytes..... | 4-88 |
| 4.1.8 | The Third Source Operand (Immediate Byte)..... | 4-88 |
| 4.1.9 | AVX Instructions and the Upper 128-bits of YMM registers | 4-89 |
| 4.1.10 | AVX Instruction Length | 4-89 |

CHAPTER 5 INSTRUCTION SET REFERENCE

| | | |
|-------|---|-----|
| 5.1 | Interpreting Instruction Reference Pages | 5-1 |
| 5.1.1 | Instruction Format | 5-1 |
| | VPERMIL2PS- Permute 2 Source 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..... | 5-5 |

| | | |
|---------|--|-------|
| 5.2 | AES Transformations and Data Structure | 5-6 |
| 5.2.1 | Little-Endian Architecture and Big-Endian Specification (FIPS 197) | 5-6 |
| 5.2.1.1 | AES Data Structure in Intel 64 Architecture | 5-6 |
| 5.2.2 | AES Transformations and Functions | 5-8 |
| 5.3 | Summary of Terms | 5-12 |
| 5.4 | Instruction SET Reference | 5-13 |
| | ADDPD - Add Packed Double Precision Floating-Point Values | 5-14 |
| | ADDPS- Add Packed Single Precision Floating-Point Values | 5-16 |
| | ADDSD- Add Scalar Double Precision Floating-Point Values | 5-18 |
| | ADDSS- Add Scalar Single Precision Floating-Point Values | 5-20 |
| | ADDSUBPD- Packed Double FP Add/Subtract | 5-22 |
| | ADDSUBPS- Packed Single FP Add/Subtract | 5-24 |
| | AEENC/AEENCLAST- Perform One Round of an AES Encryption Flow | 5-26 |
| | AESDEC/AESDECLAST- Perform One Round of an AES Decryption Flow | 5-29 |
| | AESIMC- Perform the AES InvMixColumn Transformation | 5-32 |
| | AESKEYGENASSIST - AES Round Key Generation Assist | 5-34 |
| | ANDPD- Bitwise Logical AND of Packed Double Precision Floating-Point Values | 5-36 |
| | ANDPS- Bitwise Logical AND of Packed Single Precision Floating-Point Values | 5-38 |
| | ANDNPD- Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values | 5-40 |
| | ANDNPS- Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values | 5-42 |
| | BLENDPD- Blend Packed Double Precision Floating-Point Values | 5-44 |
| | BLENDPS- Blend Packed Single Precision Floating-Point Values | 5-46 |
| | BLENDVPD- Blend Packed Double Precision Floating-Point Values | 5-49 |
| | BLENDVPS- Blend Packed Single Precision Floating-Point Values | 5-52 |
| | VBROADCAST- Load with Broadcast | 5-55 |
| | CMPPD- Compare Packed Double-Precision Floating-Point Values | 5-59 |
| | CMPPS- Compare Packed Single-Precision Floating-Point Values | 5-67 |
| | CMPSD- Compare Scalar Double-Precision Floating-Point Values | 5-74 |
| | CMPSS- Compare Scalar Single-Precision Floating-Point Values | 5-79 |
| | COMISD- Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS | 5-84 |
| | COMISS- Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS | 5-86 |
| | CVTDQ2PD- Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values | 5-88 |
| | CVTDQ2PS- Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values | 5-90 |
| | CVTPD2DQ- Convert Packed Double-Precision Floating-point values to Packed Doubleword Integers | 5-92 |
| | CVTPD2PS- Convert Packed Double-Precision Floating-point values to Packed Single-Precision Floating-Point Values | 5-95 |
| | CVTPS2DQ- Convert Packed Single Precision Floating-Point Values to Packed Singed Doubleword Integer Values | 5-98 |
| | CVTPS2PD- Convert Packed Single Precision Floating-point values to Packed Double Precision Floating-Point Values | 5-100 |
| | CVTSD2SI- Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer | |

| | |
|---------------|---|
| 5-103 | |
| CVTSD2SS- | Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value5-105 |
| CVTSI2SD- | Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value 5-107 |
| CVTSI2SS- | Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value. 5-109 |
| CVTSS2SD- | Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value5-111 |
| CVTSS2SI- | Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer. 5-113 |
| CVTTPD2DQ- | Convert with Truncation Packed Double-Precision Floating-point values to Packed Doubleword Integers5-115 |
| CVTTPS2DQ- | Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Doubleword Integer Values5-118 |
| CVTTS2SI- | Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer.5-120 |
| CVTSS2SI- | Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer.5-122 |
| DIVPD- | Divide Packed Double-Precision Floating-Point Values5-124 |
| DIVPS- | Divide Packed Single-Precision Floating-Point Values5-126 |
| DIVSD- | Divide Scalar Double-Precision Floating-Point Values.5-128 |
| DIVSS- | Divide Scalar Single-Precision Floating-Point Values.5-130 |
| DPPD- | Dot Product of Packed Double-Precision Floating-Point Values.5-132 |
| DPPS- | Dot Product of Packed Single-Precision Floating-Point Values.5-134 |
| VEXTRACTF128- | Extract packed floating-point values5-137 |
| EXTRACTPS- | Extract packed floating-point values5-139 |
| HADDPD- | Add Horizontal Double Precision Floating-Point Values5-141 |
| HADDPs- | Add Horizontal Single Precision Floating-Point Values5-143 |
| HsUBPD- | Subtract Horizontal Double Precision Floating-Point Values5-146 |
| HsUBPS- | Subtract Horizontal Single Precision Floating-Point Values5-148 |
| VINSERTF128- | Insert packed floating-point values.5-151 |
| INSERTPS- | Insert Scalar Single Precision Floating-Point Value5-152 |
| LDDQU- | Move Unaligned Integer5-156 |
| VLDMXCSR | —Load MXCSR Register5-158 |
| MASKMOVDQU- | Store Selected Bytes of Double Quadword with NT Hint.5-159 |
| VMaskMOV- | Conditional SIMD Packed Loads and Stores5-161 |
| MAXPD- | Maximum of Packed Double Precision Floating-Point Values5-165 |
| MAXPS- | Minimum of Packed Single Precision Floating-Point Values5-167 |
| MAXSD- | Return Maximum Scalar Double-Precision Floating-Point Value5-170 |
| MAXSS- | Return Maximum Scalar Single-Precision Floating-Point Value5-172 |
| MINPD- | Minimum of Packed Double Precision Floating-Point Values5-174 |
| MINPS- | Minimum of Packed Single Precision Floating-Point Values5-176 |
| MINSD- | Return Minimum Scalar Double-Precision Floating-Point Value5-179 |
| MINSS- | Return Minimum Scalar Single-Precision Floating-Point Value5-181 |
| MOVAPD- | Move Aligned Packed Double-Precision Floating-Point Values.5-183 |
| MOVAPS- | Move Aligned Packed Single-Precision Floating-Point Values.5-186 |

| | |
|--|-------|
| MOVD/MOVQ- Move Doubleword and Quadword | 5-189 |
| MOVQ- Move Quadword | 5-192 |
| MOVDDUP- Replicate Double FP Values | 5-194 |
| MOVDDQA- Move Aligned Packed Integer Values | 5-196 |
| MOVDDQU- Move Unaligned Packed Integer Values | 5-198 |
| MOVHLPS - Move Packed Single-Precision Floating-Point Values High to Low | 5-200 |
| MOVHPD- Move High Packed Double-Precision Floating-Point Values | 5-202 |
| MOVHPS- Move High Packed Single-Precision Floating-Point Values | 5-204 |
| MOVLHPS - Move Packed Single-Precision Floating-Point Values Low to High | 5-206 |
| MOVLPD- Move Low Packed Double-Precision Floating-Point Values | 5-208 |
| MOVLPS- Move Low Packed Single-Precision Floating-Point Values | 5-210 |
| MOVMSKPD- Extract Double-Precision Floating-Point Sign mask | 5-212 |
| MOVMSKPS- Extract Single-Precision Floating-Point Sign mask | 5-214 |
| MOVNTDQ- Store Packed Integers Using Non-Temporal Hint | 5-216 |
| MOVNTDQA- Load Double Quadword Non-Temporal Aligned Hint | 5-218 |
| MOVNTPD- Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint | 5-221 |
| MOVNTPS- Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint | 5-223 |
| MOVSD- Move or Merge Scalar Double-Precision Floating-Point Value | 5-225 |
| MOVSHDUP- Replicate Single FP Values | 5-228 |
| MOVSLDUP- Replicate Single FP Values | 5-231 |
| MOVSS- Move or Merge Scalar Single-Precision Floating-Point Value | 5-234 |
| MOVUPD- Move Unaligned Packed Double-Precision Floating-Point Values | 5-237 |
| MOVUPS- Move Unaligned Packed Single-Precision Floating-Point Values | 5-240 |
| MPSADBW - Multiple Sum of Absolute Differences | 5-243 |
| MULPD- Multiply Packed Double Precision Floating-Point Values | 5-248 |
| MULPS- Multiply Packed Single Precision Floating-Point Values | 5-250 |
| MULSD- Multiply Scalar Double-Precision Floating-Point Values | 5-252 |
| MULSS- Multiply Scalar Single-Precision Floating-Point Values | 5-254 |
| ORPD- Bitwise Logical OR of Packed Double Precision Floating-Point Values | 5-256 |
| ORPS- Bitwise Logical OR of Packed Single Precision Floating-Point Values | 5-258 |
| PABSB/PABSW/PABSD - Packed Absolute Value | 5-260 |
| PACKSSWB/PACKSSDW- Pack with Signed Saturation | 5-263 |
| PACKUSWB/PACKUSDW- Pack with Unsigned Saturation | 5-267 |
| PADDB/PADDW/PADDD/PADDQ- Add Packed Integers | 5-270 |
| PADDDB/PADDSDW- Add Packed Signed Integers with Signed Saturation | 5-274 |
| PADDUSB/PADDUSW- Add Packed Unsigned Integers with Unsigned Saturation .. | 5-276 |
| PALIGNR - Byte Align | 5-278 |
| PAND- Logical AND | 5-280 |
| PANDN- Logical AND NOT | 5-282 |
| PAVGB/PAVGW - Average Packed Integers | 5-284 |
| PBLENDVB - Variable Blend Packed Bytes | 5-286 |
| PBLENDW - Blend Packed Words | 5-289 |
| PCLMULQDQ - Carry-Less Multiplication Quadword | 5-291 |
| PCMPESTRI - Packed Compare Explicit Length Strings, Return Index | 5-295 |
| PCMPESTRM - Packed Compare Explicit Length Strings, Return Mask | 5-297 |

| | |
|---|-------|
| PCMPISTRI - Packed Compare Implicit Length Strings, Return Index | 5-299 |
| PCMPISTRM - Packed Compare Implicit Length Strings, Return Mask | 5-301 |
| PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ- Compare Packed Integers for Equality ... | 5-303 |
| PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ- Compare Packed Integers for Greater Than | 5-307 |
| VPERMILPD- Permute Double-Precision Floating-Point Values | 5-311 |
| VPERMIL2PD- Permute 2 Source Double-Precision Floating-Point Values..... | 5-315 |
| VPERMILPS- Permute Single-Precision Floating-Point Values | 5-320 |
| VPERMIL2PS- Permute 2 Source Single-Precision Floating-Point Values..... | 5-324 |
| VPERM2F128- Permute Floating-Point Values..... | 5-329 |
| PEXTRB/PEXTRW/PEXTRD/PEXTRQ- Extract Integer | 5-331 |
| PHADDW/PHADDD - Packed Horizontal Add | 5-335 |
| PHADDSW - Packed Horizontal Add with Saturation..... | 5-337 |
| PHMINPOSUW - Horizontal Minimum and Position | 5-339 |
| PHSUBW/PHSUBD - Packed Horizontal Subtract | 5-341 |
| PHSUBSW - Packed Horizontal Subtract with Saturation | 5-343 |
| PINSRB/PINSRW/PINSRD/PINSRQ- Insert Integer | 5-345 |
| PMADDWD- Multiply and Add Packed Integers | 5-349 |
| PMADDUBSW- Multiply and Add Packed Integers..... | 5-351 |
| PMASXB/PMASXW/PMASXD- Maximum of Packed Signed Integers | 5-353 |
| PMASUB/PMASUW/PMASUD- Maximum of Packed Unsigned Integers | 5-357 |
| PMINSB/PMINSW/PMINSD- Minimum of Packed Signed Integers..... | 5-361 |
| PMINUB/PMINUW/PMINUD- Minimum of Packed Unsigned Integers | 5-365 |
| PMOVMSKB- Move Byte Mask | 5-369 |
| PMOVSW - Packed Move with Sign Extend | 5-371 |
| PMOVZX - Packed Move with Zero Extend | 5-376 |
| PMULHUW - Multiply Packed Unsigned Integers and Store High Result | 5-381 |
| PMULHRW - Multiply Packed Unsigned Integers with Round and Shift..... | 5-383 |
| PMULHW - Multiply Packed Integers and Store High Result | 5-385 |
| PMULLW/PMULLD - Multiply Packed Integers and Store Low Result..... | 5-387 |
| PMULUDQ - Multiply Packed Unsigned Doubleword Integers..... | 5-390 |
| PMULDQ - Multiply Packed Doubleword Integers..... | 5-392 |
| POR - Bitwise Logical Or..... | 5-394 |
| PSADBW - Compute Sum of Absolute Differences..... | 5-396 |
| PSHUFB - Packed Shuffle Bytes..... | 5-398 |
| PSHUFD - Shuffle Packed Doublewords | 5-400 |
| PSHUFHW - Shuffle Packed High Words | 5-402 |
| PSHUFLLW - Shuffle Packed Low Words..... | 5-404 |
| PSIGNB/PSIGNW/PSIGND - Packed SIGN | 5-406 |
| PSLLDQ - Byte Shift Left | 5-410 |
| PSRLDQ - Byte Shift Right..... | 5-412 |
| PSLLW/PSLLD/PSLLQ - Bit Shift Left | 5-414 |
| PSRAW/PSRAD - Bit Shift Arithmetic Right | 5-419 |
| PSRLW/PSRLD/PSRLQ - Shift Packed Data Right Logical..... | 5-423 |
| PTEST- Packed Bit Test | 5-428 |
| PSUBB/PSUBW/PSUBD/PSUBQ -Packed Integer Subtract..... | 5-432 |

| | |
|--|-------|
| PSUBSB/PSUBSW -Subtract Packed Signed Integers with Signed Saturation | 5-436 |
| PSUBUSB/PSUBUSW -Subtract Packed Unsigned Integers with Unsigned Saturation . | 5-438 |
| PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ - Unpack High Data | 5-440 |
| PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ - Unpack Low Data | 5-444 |
| PXOR - Exclusive Or | 5-448 |
| RCPPS- Compute Approximate Reciprocals of Packed Single-Precision Floating-Point Values | 5-450 |
| RCPSS - Compute Reciprocal of Scalar Single-Precision Floating-Point Value | 5-453 |
| RSQRTPS - Compute Approximate Reciprocals of Square Roots of Packed Single-Precision Floating-point Values | 5-455 |
| RSQRTSS - Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value | 5-458 |
| ROUNDPD- Round Packed Double-Precision Floating-Point Values | 5-460 |
| ROUNDPS- Round Packed Single-Precision Floating-Point Values | 5-464 |
| ROUNDSD - Round Scalar Double-Precision Value | 5-467 |
| ROUNDSS - Round Scalar Single-Precision Value | 5-469 |
| SHUFPD - Shuffle Packed Double Precision Floating-Point Values | 5-471 |
| SHUFPS - Shuffle Packed Single Precision Floating-Point Values | 5-474 |
| SQRTPD- Square Root of Double-Precision Floating-Point Values | 5-477 |
| SQRTPS- Square Root of Single-Precision Floating-Point Values | 5-479 |
| SQRTSD - Compute Square Root of Scalar Double-Precision Floating-Point Value . | 5-481 |
| SQRTSS - Compute Square Root of Scalar Single-Precision Value | 5-483 |
| VSTMXCSR—Store MXCSR Register State | 5-485 |
| SUBPD- Subtract Packed Double Precision Floating-Point Values | 5-486 |
| SUBPS- Subtract Packed Single Precision Floating-Point Values | 5-488 |
| SUBSD- Subtract Scalar Double Precision Floating-Point Values | 5-490 |
| SUBSS- Subtract Scalar Single Precision Floating-Point Values | 5-492 |
| UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS | 5-494 |
| UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS | 5-496 |
| UNPCKHPD- Unpack and Interleave High Packed Double-Precision Floating-Point Values | 5-498 |
| UNPCKHPS- Unpack and Interleave High Packed Single-Precision Floating-Point Values | 5-500 |
| UNPCKLPD- Unpack and Interleave Low Packed Double-Precision Floating-Point Values | 5-503 |
| UNPCKLPS- Unpack and Interleave Low Packed Single-Precision Floating-Point Values . | 5-505 |
| XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values . . | 5-508 |
| XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values . . | 5-510 |
| VZEROALL- Zero All YMM registers | 5-512 |
| VZERoupper- Zero Upper bits of YMM registers | 5-514 |

CHAPTER 6
INSTRUCTION SET REFERENCE - FMA

6.1 FMA Instruction SET Reference 6-1

- VFMADDPD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values 6-2
- VFMADDPS - Fused Multiply-Add of Packed Single-Precision Floating-Point Values 6-6
- VFMADDSD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values 6-9
- VFMADDSS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 6-11
- VFMADDSBPD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values 6-13
- VFMADDSBPS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values 6-16
- VFMSUBDDPD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values 6-19
- VFMSUBDDPS - Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values 6-22
- VFMSUBDPD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values 6-25
- VFMSUBPS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values 6-28
- VFMSUBSD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values 6-31
- VFMSUBSS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 6-33
- VFMNADDPD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 6-35
- VFMNADDPS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values 6-38
- VFMNADDSD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values 6-41
- VFMNADDSS - Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values 6-43
- VFNMSBPD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values 6-45
- VFNMSBPS - Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values 6-48
- VFNMSBPD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values 6-51
- VFNMSBSS - Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values 6-53

APPENDIX A
INSTRUCTION SUMMARY

APPENDIX B
INSTRUCTION OPCODE MAP

INDEX

TABLES

| | | |
|------------|---|------|
| Table 2-1 | Rounding behavior of Zero Result in FMA Operation | 2-13 |
| Table 2-2 | FMA Numeric Behavior | 2-14 |
| Table 2-3 | Alignment Faulting Conditions when Memory Access is Not Aligned | 2-17 |
| Table 2-4 | Instructions Requiring Explicitly Aligned Memory | 2-17 |
| Table 2-5 | Instructions Not Requiring Explicit Memory Alignment | 2-18 |
| Table 2-6 | Exception class description | 2-19 |
| Table 2-7 | Instructions in each Exception Class | 2-20 |
| Table 2-8 | #UD Exception and VEX.L Field Encoding | 2-22 |
| Table 2-9 | Type 1 Class Exception Conditions | 2-23 |
| Table 2-10 | Type 2 Class Exception Conditions | 2-25 |
| Table 2-11 | Type 3 Class Exception Conditions | 2-26 |
| Table 2-12 | Type 4 Class Exception Conditions | 2-27 |
| Table 2-13 | Type 5 Class Exception Conditions | 2-28 |
| Table 2-14 | Type 6 Class Exception Conditions | 2-29 |
| Table 2-15 | Type 7 Class Exception Conditions | 2-30 |
| Table 2-16 | Type 8 Class Exception Conditions | 2-30 |
| Table 2-17 | Type 9 Class Exception Conditions | 2-31 |
| Table 2-18 | Information Returned by CPUID Instruction | 2-35 |
| Table 2-19 | Highest CPUID Source Operand for Intel 64 and IA-32 Processors | 2-44 |
| Table 2-20 | Processor Type Field | 2-45 |
| Table 2-21 | Feature Information Returned in the ECX Register | 2-47 |
| Table 2-22 | More on Feature Information Returned in the EDX Register | 2-51 |
| Table 2-23 | Encoding of Cache and TLB Descriptors | 2-53 |
| Table 2-24 | Processor Brand String Returned with Pentium 4 Processor | 2-59 |
| Table 2-25 | Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings | 2-62 |
| Table 3-1 | XFEATURE_ENABLED_MASK and Processor State Components | 3-69 |
| Table 3-2 | CR4 bits for AVX New Instructions technology support | 3-69 |
| Table 3-3 | Layout of XSAVE Area For Processor Supporting YMM State | 3-70 |
| Table 3-4 | XSAVE Header Format | 3-70 |
| Table 3-5 | XSAVE Save Area Layout for YMM State (Ext_Save_Area_2) | 3-71 |
| Table 3-6 | XRSTOR Action on MXCSR, XMM Registers, YMM Registers | 3-71 |
| Table 3-7 | Processor Supplied Init Values XRSTOR May Use | 3-72 |
| Table 3-8 | XSAVE Action on MXCSR, XMM, YMM Register | 3-72 |
| Table 4-1 | VEX.vvvv to register name mapping | 4-82 |
| Table 4-2 | Instructions with a VEX.vvvv destination | 4-83 |
| Table 4-3 | Interpreting VEX.vvvv, reg_field, and rm_field. | 4-84 |
| Table 4-4 | VEX.m-mmmm interpretation | 4-85 |
| Table 4-5 | VEX.L interpretation | 4-86 |
| Table 4-6 | VEX.pp interpretation | 4-86 |
| Table 5-1 | Byte and 32-bit Word Representation of a 128-bit State | 5-7 |
| Table 5-2 | Matrix Representation of a 128-bit State | 5-7 |
| Table 5-3 | Little Endian Representation of a 128-bit State | 5-7 |
| Table 5-4 | Little Endian Representation of a 4x4 Byte Matrix | 5-7 |
| Table 5-5 | The ShiftRows Transformation | 5-9 |

| | | |
|------------|---|-------|
| Table 5-6 | Look-up Table Associated with S-Box Transformation | 5-10 |
| Table 5-7 | The InverseShiftRows Transformation | 5-11 |
| Table 5-8 | Look-up Table Associated with InvS-Box Transformation | 5-12 |
| Table 5-9 | Comparison Predicate for CMPPD and CMPPS Instructions | 5-57 |
| Table 5-10 | Pseudo-Op and CMPPD Implementation | 5-60 |
| Table 5-11 | Pseudo-Op and VCMPPD Implementation | 5-60 |
| Table 5-12 | Pseudo-Op and CMPPS Implementation | 5-65 |
| Table 5-13 | Pseudo-Op and VCMPPS Implementation | 5-66 |
| Table 5-14 | Pseudo-Op and CMPSD Implementation | 5-72 |
| Table 5-15 | Pseudo-Op and VCMPSD Implementation | 5-73 |
| Table 5-16 | Pseudo-Op and CMPSS Implementation | 5-77 |
| Table 5-17 | Pseudo-Op and VCMPS Implementation | 5-78 |
| Table 5-18 | PCLMULQDQ Quadword Selection of Immediate Byte | 5-287 |
| Table 5-19 | VPERMIL2PD Action and ZERO_MATCH Encoding | 5-312 |
| Table 5-20 | Pseudo-Op and VPERMIL2PD Assembly Support for VEX.W = 0 | 5-313 |
| Table 5-21 | VPERMIL2PS Action and ZERO_MATCH Encoding | 5-321 |
| Table 5-22 | Pseudo-Op and VPERMIL2PS Assembly Support for VEX.W = 0 | 5-322 |
| Table A-1 | Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions | A-2 |
| Table A-2 | AVX, FMA and AES New Instructions | A-10 |

FIGURES

- Figure 2-1 General Procedural Flow of Application Detection of AVX 2-10
- Figure 2-2 Version Information Returned by CPUID in EAX 2-45
- Figure 2-3 Feature Information Returned in the ECX Register 2-47
- Figure 2-4 Feature Information Returned in the EDX Register 2-50
- Figure 2-5 Determination of Support for the Processor Brand String 2-59
- Figure 2-6 Algorithm for Extracting Maximum Processor Frequency 2-61
- Figure 4-1 Instruction Encoding Format with VEX Prefix 4-76
- Figure 4-2 VEX bitfields 4-79
- Figure 5-1 VBROADCASTSS Operation (VEX.256 encoded version) 5-53
- Figure 5-2 VBROADCASTSS Operation (128-bit version) 5-53
- Figure 5-3 VBROADCASTSD Operation 5-54
- Figure 5-4 VBROADCASTF128 Operation 5-54
- Figure 5-5 CVTQ2PD (VEX.256 encoded version) 5-86
- Figure 5-6 VCVTPD2DQ (VEX.256 encoded version) 5-90
- Figure 5-7 VCVTPD2PS (VEX.256 encoded version) 5-93
- Figure 5-8 CVTPS2PD (VEX.256 encoded version) 5-98
- Figure 5-9 VCVTPD2DQ (VEX.256 encoded version) 5-113
- Figure 5-10 VHADDPD operation 5-138
- Figure 5-11 VHADDPS operation 5-140
- Figure 5-12 VHSUBPD operation 5-143
- Figure 5-13 VHSUBPS operation 5-145
- Figure 5-14 VMOVDDUP Operation 5-192
- Figure 5-15 MOVSHDUP Operation 5-225
- Figure 5-16 MOVSLDUP Operation 5-228
- Figure 5-17 PACKSSDw Instruction Operation using 64-bit Operands 5-260
- Figure 5-18 VPERMILPD operation 5-307
- Figure 5-19 VPERMILPD Shuffle Control 5-307
- Figure 5-20 VPERMIL2PD Control Field 5-311
- Figure 5-21 VPERMIL2PD Immediate Byte 5-312
- Figure 5-22 VPERMIL2PD operation 5-312
- Figure 5-23 VPERMILPS Operation 5-316
- Figure 5-24 VPERMILPS Shuffle Control 5-316
- Figure 5-25 VPERMIL2PS Control Field 5-320
- Figure 5-26 VPERMIL2PS Immediate Byte 5-321
- Figure 5-27 VPERMIL2PS operation 5-322
- Figure 5-28 VPERM2F128 Operation 5-325
- Figure 5-29 PSHUFD Instruction Operation 5-396
- Figure 5-30 PUNPCKHDQ Instruction Operation 5-437
- Figure 5-31 PUNPCKLBW Instruction Operation using 64-bit Operands 5-441
- Figure 5-32 VROUNDPD immediate control field definition 5-457
- Figure 5-33 VSHUFPD Operation 5-468
- Figure 5-34 VSHUFPS Operation 5-471
- Figure 5-35 VUNPCKHPS Operation 5-497
- Figure 5-36 VUNPCKLPS Operation 5-502
- Figure 6-1 VFMADDPD Immediate Byte 6-3

CHAPTER 1

INTEL® ADVANCED VECTOR EXTENSIONS

1.1 ABOUT THIS DOCUMENT

This document describes the software programming interfaces of several vector SIMD instruction extensions of the Intel® 64 architecture that will be introduced starting with Intel 64 processors built on 32nm process technology. The instruction set extensions covered in this document, with respect to availability in different processor generations, is referred to by the following categories:

- General-purpose encryption and AES: 128-bit SIMD extensions targeted to accelerate high-speed block encryption and cryptographic processing using the Advanced Encryption Standard.
- Intel® Advanced Vector Extensions (AVX) introduces 256-bit vector processing capability and includes two components to be introduced on Intel processor generations built from 32nm process and beyond:
 - The first generation Intel AVX provides 256-bit SIMD register support, 256-bit vector floating-point instructions, enhancements to 128-bit SIMD instructions, support for three and four operand syntax.
 - FMA is a future extension of Intel AVX, FMA provides floating-point, fused multiply-add instructions supporting 256-bit and 128-bit SIMD vectors.

Chapter 1 provides an overview of these instruction set extensions. Chapter 2 describes the application programming environment. Chapter 3 describes system programming requirements needed to support 256-bit registers. Chapter 4 describes the architectural extensions of Intel 64 instruction encoding format that support 256-bit registers, three and four operand syntax. Chapter 5 provides detailed instruction reference for AVX and encryption/AES instructions. Chapter 6 provides detailed instruction reference for FMA instructions.

1.2 OVERVIEW

Intel® Advanced Vector Extensions extend beyond the capabilities and programming environment over those of multiple generations of Streaming SIMD Extensions. Intel AVX address the continued need for vector floating-point performance in mainstream scientific and engineering numerical applications, visual processing, recognition, data-mining/synthesis, gaming, physics, cryptography and other areas of applications. Intel AVX is designed to facilitate efficient implementation by wide spectrum of software architectures of varying degrees of thread parallelism, and data vector lengths. Intel AVX offers the following benefits:

- efficient building blocks for applications targeted across all segments of computing platforms.

- significant increase in floating-point performance density with good power efficiency over previous generations of 128-bit SIMD instruction set extensions,
- scalable performance with multi-core processor capability.

Intel AVX also establishes a foundation for future evolution in both instruction set functionality and vector lengths by introducing an efficient instruction encoding scheme, three and four operand instruction syntax, supporting load and store masking, etc.

Intel Advanced Vector Extensions offers comprehensive architectural enhancements and functional enhancements in arithmetic as well as data processing primitives. Section 1.3 summarizes the architectural enhancement of AVX. Functional overview of AVX and FMA instructions are summarized in Section 1.4. General-purpose encryption and AES instructions follow the existing architecture of 128-bit SIMD instruction sets like SSE4 and its predecessors, Section 1.5 provides a short summary.

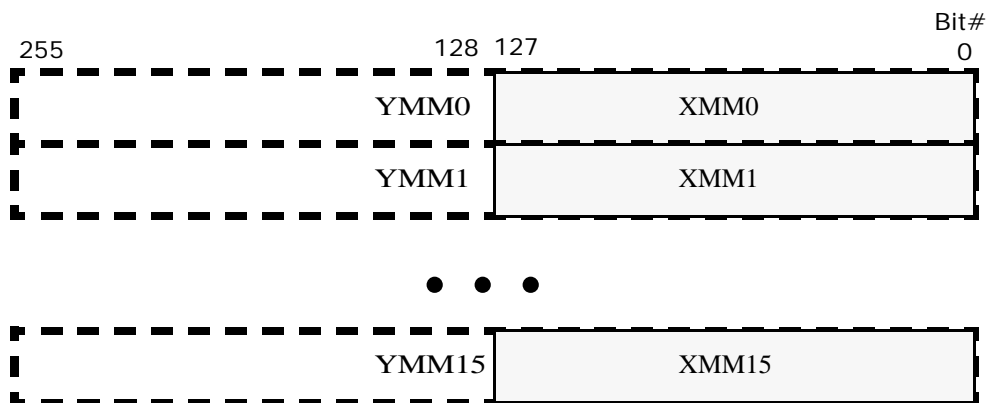
1.3 INTEL® ADVANCED VECTOR EXTENSIONS ARCHITECTURE OVERVIEW

Intel AVX has many similarities to the SSE and double-precision floating-point portions of SSE2. However, Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set. 256-bit register state is managed by Operating System using XSAVE/XRSTOR instructions introduced in 45 nm Intel 64 processors (see *IA-32 Intel® Architecture Software Developer's Manual, Volumes 2B and 3A*).
- Instruction syntax support for three and four operands to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Instruction encoding format using a new prefix (referred to as VEX) to provide compact, efficient encoding for three-operand syntax, vector lengths, compaction of SIMD prefixes and REX functionality.

1.3.1 256-Bit Wide SIMD Register Support

Intel AVX introduces support for 256-bit wide SIMD registers (YMM0-YMM7 in operating modes that are 32-bit or less, YMM0-YMM15 in 64-bit mode). The lower 128-bits of the YMM registers are aliased to the respective 128-bit XMM registers.



1.3.2 Instruction Syntax Enhancements

Intel AVX employs an instruction encoding scheme using a new prefix (known as “VEX” prefix). Instruction encoding using the VEX prefix can directly encode a register operand within the VEX prefix. This support two new instruction syntax in Intel 64 architecture:

- A non-destructive operand (in a three-operand instruction syntax): The non-destructive source reduces the number of registers, register-register copies and explicit load operations required in typical SSE loops, reduces code size, and improves micro-fusion opportunities.
- A third source operand (in a four-operand instruction syntax) via the upper 4 bits in an 8-bit immediate field. Support for the third source operand is defined for selected instructions (e.g. FMA, VBLENDVPD, VPERMIL2PD, etc.).

Two-operand instruction syntax previously expressed as

```
ADDPS xmm1, xmm2/m128
```

now can be expressed in three-operand syntax as

```
VADDPS xmm1, xmm2, xmm3/m128
```

In four-operand syntax, the extra register operand is encoded in the immediate byte.

Note SIMD instructions supporting three-operand syntax but processing only 128-bits of data are considered part of the 256-bit SIMD instruction set extensions of AVX, because bits 255:128 of the destination register are zeroed by the processor.

1.3.3 VEX Prefix Instruction Encoding Support

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

- Direct encoding of a register operand within VEX. This provides instruction syntax support for non-destructive source operand.
- Efficient encoding of instruction syntax operating on 128-bit and 256-bit register sets.
- Compaction of REX prefix functionality: The equivalent functionality of the REX prefix is encoded within VEX.
- 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 replaced by a more compact representation of opcode extension within the VEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the VEX prefix encoding.
- Source operand swizzling functionality: VEX prefix can provide encoding capability to support source register operand swizzling. This generally applies to instruction syntax with four operands.
- Most VEX-encoded SIMD numeric and data processing instruction semantics with memory operand have relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 2.5).

VEX prefix encoding applies to SIMD instructions operating on YMM registers, XMM registers, and in some cases with a general-purpose register as one of the operand. VEX prefix is not supported for instructions operating on MMX or x87 registers. Details of VEX prefix and instruction encoding are discussed in Chapter 4.

1.4 FUNCTIONAL OVERVIEW

Intel AVX and FMA provide comprehensive functional improvements over previous generations of SIMD instruction extensions. The functional improvements include:

- 256-bit floating-point arithmetic primitives: AVX enhances existing 128-bit floating-point arithmetic instructions with 256-bit capabilities for floating-point processing. FMA provides additional set of 256-bit floating-point processing capabilities with a rich set of fused-multiply-add and fused multiply-subtract primitives.

- Enhancements for flexible SIMD data movements: AVX provides a number of new data movement primitives to enable efficient SIMD programming in relation to loading non-unit-strided data into SIMD registers, intra-register SIMD data manipulation, conditional expression and branch handling, etc. Enhancements for SIMD data movement primitives cover 256-bit and 128-bit vector floating-point data, and across 128-bit integer SIMD data processing using VEX-encoded instructions.

Several key categories of functional improvements in AVX and FMA are summarized in the following subsections.

1.4.1 256-bit Floating-Point Arithmetic Processing Enhancements

Intel AVX provides 35 256-bit floating-point arithmetic instructions. The arithmetic operations cover add, subtract, multiply, divide, square-root, compare, max, min, round, etc., on single-precision and double-precision floating-point data.

The enhancement in AVX on floating-point compare operation provides 32 conditional predicates to improve programming flexibility in evaluating conditional expressions.

FMA provides 12 256-bit floating-point instructions to perform computation on 256-bit vectors. The arithmetic operations cover fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract.

1.4.2 256-bit Non-Arithmetic Instruction Enhancements

Intel AVX provides new primitives for handling data movement within 256-bit floating-point vectors and promotes many 128-bit floating data processing instructions to handle 256-bit floating-point vectors.

AVX includes 37 256-bit data processing instructions that are promoted from previous generations of SIMD instruction extensions, ranging from logical, blend, convert, test, unpacking, shuffling, load and stores.

AVX introduces 21 new data processing instructions that operate on 256-bit vectors. These new primitives cover the following operations:

- Non-unit-strided fetching of SIMD data. AVX provides several flexible SIMD floating-point data fetching primitives:
 - broadcast of single or multiple data elements into a 256-bit destination,
 - masked move primitives to load or store SIMD data elements conditionally,
- Intra-register manipulation of SIMD data elements. AVX provides several flexible SIMD floating-point data manipulation primitives:
 - insert/extract multiple SIMD floating-point data elements to/from 256-bit SIMD registers

- permute primitives to facilitate efficient manipulation of floating-point data elements in 256-bit SIMD registers
- Branch handling. AVX provides several primitives to enable handling of branches in SIMD programming:
 - new variable blend instructions supports four-operand syntax with non-destructive source syntax. This is more flexible than the equivalent SSE4 instruction syntax which uses the XMM0 register as the implied mask for blend selection.
 - Packed TEST instructions for floating-point data.

1.4.3 Arithmetic Primitives for 128-bit Vector and Scalar processing

Intel AVX provides 131 128-bit numeric processing instructions that employ VEX-prefix encoding. These VEX-encoded instructions generally provide the same functionality over instructions operating on XMM register that are encoded using SIMD prefixes. The 128-bit numeric processing instructions in AVX cover floating-point and integer data processing; across 128-bit vector and scalar processing.

The enhancement in AVX on 128-bit floating-point compare operation provides 32 conditional predicates to improve programming flexibility in evaluating conditional expressions. This contrasts with floating-point SIMD compare instructions in SSE and SSE2 supporting only 8 conditional predicates.

FMA provides 20 128-bit floating-point instructions to process 128-bit vector and scalar data. The arithmetic operations cover fused multiply-add, fused multiply-subtract, signed-reversed multiply on fused multiply-add and multiply-subtract.

1.4.4 Non-Arithmetic Primitives for 128-bit Vector and Scalar Processing

Intel AVX provides 126 data processing instructions that employ VEX-prefix encoding. These VEX-encoded instructions generally provide the same functionality over instructions operating on XMM register that are encoded using SIMD prefixes. The 128-bit data processing instructions in AVX cover floating-point and integer data movement primitives.

Additional enhancements in AVX on 128-bit data processing primitives include 16 new instructions with the following capabilities:

- Non-unit-strided fetching of SIMD data. AVX provides several flexible SIMD floating-point data fetching primitives:
 - broadcast of single data element into a 128-bit destination,
 - masked move primitives to load or store SIMD data elements conditionally,
- Intra-register manipulation of SIMD data elements. AVX provides several flexible SIMD floating-point data manipulation primitives:

- permute primitives to facilitate efficient manipulation of floating-point data elements in 128-bit SIMD registers
- Branch handling. AVX provides several primitives to enable handling of branches in SIMD programming:
 - new variable blend instructions supports four-operand syntax with non-destructive source syntax. Branching conditions dependent on floating-point data or integer data can benefit from Intel AVX. This is more flexible than non-VEX encoded instruction syntax that uses the XMM0 register as implied mask for blend selection. While variable blend with implied XMM0 syntax is supported in SSE4 using SIMD prefix encoding, VEX-encoded 128-bit variable blend instructions only support the more flexible four-operand syntax.
 - Packed TEST instructions for floating-point data.

1.5 GENERAL ENCRYPTION AND CRYPTOGRAPHIC PROCESSING

Intel 64 processors using 32nm processing technology introduces several primitives targeted to accelerate general-purpose block encryption and cryptographic functions using the Advanced Encryption Standard (AES) of block cipher encryption and decryption on 128-bit blocks.

General-purpose block encryption primitives are provided by PCLMULQDQ instruction, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

AES encryption involves processing 128-bit input data (plaintext) through a finite number of iterative operation, referred to as “AES round”, into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the “equivalent inverse cipher” instead of the “inverse cipher”.

The cryptographic processing at each round involves two input data, one is the “state”, the other is the “round key”. Each round uses a different “round key”. The round keys are derived from the cipher key using a “key schedule” algorithm. The “key schedule” algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES standard supports cipher key of sizes 128, 192, and 256 bits. The respective cipher key sizes correspond to 10, 12, and 14 rounds of iteration.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

This page was
intentionally left
blank.

CHAPTER 2

APPLICATION PROGRAMMING MODEL

The application programming model for Intel AVX, FMA, AES and encryption primitives extend from that of Streaming SIMD Extensions (SSE) and is summarized as follows:

- The AES extensions and carry-less multiplication instruction (PCLMULQDQ) follow the same programming model as SSE, SSE2, SSE3, SSSE3, and SSE4 (see *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The detection of AES and PCLMULQDQ is described in Section 2.1.
- The AVX and FMA extensions follow a programming model analogous to that of SSE with minor differences. This is described in Section 2.1 through Section 2.8. Note however that the OS support and detection process has changed considerably.
- The numeric exception behavior of FMA is similar to previous generations of SIMD floating-point instructions. The specific details are described in Section 2.3.

CPUID instruction details for detecting AVX, FMA, AES, PCLMULQDQ are described in Section 2.9.

2.1 DETECTION OF PCLMULQDQ AND AES INSTRUCTIONS

Before an application attempts to use the following AES instructions: AESDEC/AESDECLAST/AESENCAESENCLAST/AESIMC/AESKEYGENASSIST, it must check that the processor supports the AES extensions. AES extensions is supported if CPUID.01H:ECX.AES[bit 25] = 1.

Prior to using PCLMULQDQ instruction, application must check if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.

Operating systems that support handling SSE state will also support applications that use AES extensions and PCLMULQDQ instruction. This is the same requirement for SSE2, SSE3, SSSE3, and SSE4.

2.2 DETECTION OF AVX AND FMA INSTRUCTIONS

AVX and FMA operate on the 256-bit YMM register state. System software requirements to support YMM state is described in Chapter 3.

Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 2-1.

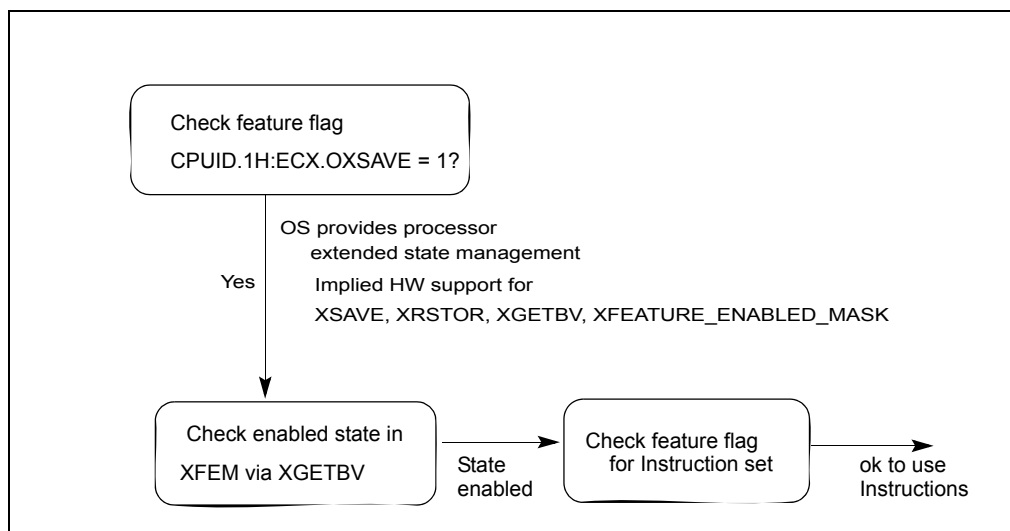


Figure 2-1. General Procedural Flow of Application Detection of AVX

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

- 1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use¹)
- 2) Issue XGETBV and verify that XFEATURE_ENABLED_MASK[2:1] = '11b' (XMM state and YMM state are enabled by OS).
- 3) detect CPUID.1:ECX.AVX[bit 28] = 1 (AVX instructions supported).

(Step 3 can be done in any order relative to 1 and 2)

The following pseudocode illustrates this recommended application AVX detection process:

INT supports_AVX()

```
{
    ; result in eax
    mov eax, 1
    cpuid
}
```

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


```

and ecx, 018000000H
cmp ecx, 018000000H; check both OSXSAVE and AVX feature flags
jne not_supported
; processor supports AVX instructions and XGETBV is enabled by OS
mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
XGETBV; result in EDX:EAX
and eax, 06H
cmp eax, 06H; check OS has enabled both XMM and YMM state support
jne not_supported
mov eax, 1
jmp done
NOT_SUPPORTED:
mov eax, 0
done:
}

```

Note: It is unwise for an application to rely exclusively on CPUID.1:ECX.AVX[bit 28] or at all on CPUID.1:ECX.XSAVE[bit 26]: These indicate hardware support but not operating system support. If YMM state management is not enabled by an operating system, AVX instructions will #UD regardless of CPUID.1:ECX.AVX[bit 28]. “CPUID.1:ECX.XSAVE[bit 26] = 1” does not guarantee the OS actually uses the XSAVE process for state management.

These steps above also apply to enhanced 128-bit SIMD floating-pointing instructions in AVX (using VEX prefix-encoding) that operate on the YMM states. Application detection of VEX-encoded AES is described in Section 2.2.2.

2.2.1 Detection of FMA

Hardware support for FMA is indicated by CPUID.1:ECX.FMA[bit 12]=1.

Application Software must identify that hardware supports AVX as explained in Section 2.2, after that it must also detect support for FMA by CPUID.1:ECX.FMA[bit 12]. The recommended pseudocode sequence for detection of FMA is:

```

INT supports_fma()
{
    ; result in eax
    mov eax, 1
    cpuid
    and ecx, 018001000H

```

```

    cmp ecx, 018001000H; check OSXSAVE, AVX, FMA feature flags
    jne not_supported
    ; processor supports AVX,FMA instructions and XGETBV is enabled by OS
    mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
    XGETBV; result in EDX:EAX
    and eax, 06H
    cmp eax, 06H; check OS has enabled both XMM and YMM state support
    jne not_supported
    mov eax, 1
    jmp done
NOT_SUPPORTED:
    mov eax, 0
done:
}

```

Note that FMA comprises of 256-bit and 128-bit SIMD instructions operating on YMM states.

2.2.2 Detection of VEX-Encoded AES

VAESDEC/VAESDECLAST/VAESEC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST instructions operate on YMM states. The detection sequence must combine checking for CPUID.1:ECX.AES[bit 25] = 1 and the sequence for detection application support for AVX.

This is shown in the pseudocode:

```

-----
INT supports_VAES()
{
    ; result in eax
    mov eax, 1
    cpuid
    and ecx, 01A000000H
    cmp ecx, 01A000000H; check OSXSAVE, AVX and AES feature flags
    jne not_supported
    ; processor supports AVX and VEX.128-encoded AES instructions and XGETBV
    is enabled by OS
    mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
    XGETBV; result in EDX:EAX
}

```

```

and eax, 06H
cmp eax, 06H; check OS has enabled both XMM and YMM state support
jne not_supported
mov eax, 1
jmp done
NOT_SUPPORTED:
mov eax, 0
done:
}

```

2.3 FUSED-MULTIPLY-ADD (FMA) NUMERIC BEHAVIOR

FMA instructions can perform fused-multiply-add operations (including fused-multiply-subtract, and other varieties) on packed and scalar data elements in the instruction operands. FMA instruction provide separate instructions to handle different types of arithmetic operations on the three source operands (for example VFMAADDSUBPD and VFMSUBADDDPD perform different types of arithmetic operations on different data elements in the three source operands).

An arithmetic FMA operation performed in an FMA instruction takes one of several forms, $r=x*y+z$, $r=x*y-z$, $r=-x*y+z$, or $r=-x*y-z$. Packed FMA instructions can perform eight single-precision FMA operations or four double-precision FMA operations with 256-bit vectors. Note scalar FMA instructions only perform one arithmetic operation on the low order data element of the three operands and the rest of the upper data elements in the destination operand is filled with zero.

An arithmetic FMA operation of the form, $r=x*y+z$, takes two IEEE-754-1985 single (double) precision values and multiplies them to form an infinite precision intermediate value. This intermediate value is added to a third single (double) precision value (also at infinite precision) and rounded to produce a single (double) precision result.

Table 2-2 describes the numerical behavior of the FMA operation, $r=x*y+z$, $r=x*y-z$, $r=-x*y+z$, $r=-x*y-z$ for various input values. The input values can be 0, finite non-zero (F in Table 2-2), infinity of either sign (INF in Table 2-2), positive infinity (+INF in Table 2-2), negative infinity (-INF in Table 2-2), or NaN (including QNaN or SNaN). If any one of the input values is a NaN, the result of FMA operation, r , may be a quietized NaN. The result can be either $Q(x)$, $Q(y)$, or $Q(z)$, see Table 2-2. If x is a NaN, then:

- $Q(x) = x$ if x is QNaN or

- $Q(x)$ = the quietized NaN obtained from x if x is SNaN

The notation for output value in Table 2-2 are:

- “+INF”: positive infinity, “-INF”: negative infinity. When the result depends on a conditional expression, both values are listed in the result column and the condition is described in the comment column.
- QNaNIndefinite represents the QNaN which has the sign bit equal to 1, the second most significant field equal to 1, and the remaining significant field bits equal to 0.
- The summation or subtraction of 0s or identical values in FMA operation can lead to the following situations shown in Table 2-1

Table 2-1. Rounding behavior of Zero Result in FMA Operation

| $x*y$ | z | $x*y + z$ | $x*y - z$ | $-x*y + z$ | $-x*y - z$ |
|-------|------|--|--|--|--|
| (+0) | (+0) | +0 in all rounding modes | - 0 when rounding down, and +0 otherwise | - 0 when rounding down, and +0 otherwise | - 0 in all rounding modes |
| (+0) | (-0) | - 0 when rounding down, and +0 otherwise | +0 in all rounding modes | - 0 in all rounding modes | - 0 when rounding down, and +0 otherwise |
| (-0) | (+0) | - 0 when rounding down, and +0 otherwise | - 0 in all rounding modes | + 0 in all rounding modes | - 0 when rounding down, and +0 otherwise |
| (-0) | (-0) | - 0 in all rounding modes | - 0 when rounding down, and +0 otherwise | - 0 when rounding down, and +0 otherwise | +0 in all rounding modes |
| F | -F | - 0 when rounding down, and +0 otherwise | $2*F$ | $-2*F$ | - 0 when rounding down, and +0 otherwise |
| F | F | $2*F$ | - 0 when rounding down, and +0 otherwise | - 0 when rounding down, and +0 otherwise | $-2*F$ |

Table 2-2. FMA Numeric Behavior

| x | y | z | $r=x*y+z$ | $r=x*y-z$ | $r = -x*y+z$ | $r = -x*y-z$ | Comment |
|-----------|----------------|----------------|----------------|----------------|----------------|----------------|---|
| NaN | 0, F, INF, NaN | 0, F, INF, NaN | Q(x) | Q(x) | Q(x) | Q(x) | Signal invalid exception if x or y or z is SNaN |
| 0, F, INF | NaN | 0, F, INF, NaN | Q(y) | Q(y) | Q(y) | Q(y) | Signal invalid exception if y or z is SNaN |
| 0, F, INF | 0, F, INF | NaN | Q(z) | Q(z) | Q(z) | Q(z) | Signal invalid exception if z is SNaN |
| INF | F, INF | +INF | +INF | QNaNIndefinite | QNaNIndefinite | -INF | if $x*y$ and z have the same sign |
| | | | QNaNIndefinite | -INF | +INF | QNaNIndefinite | if $x*y$ and z have opposite signs |
| INF | F, INF | -INF | -INF | QNaNIndefinite | QNaNIndefinite | +INF | if $x*y$ and z have the same sign |
| | | | QNaNIndefinite | +INF | -INF | QNaNIndefinite | if $x*y$ and z have opposite signs |
| INF | F, INF | 0, F | +INF | +INF | -INF | -INF | if x and y have the same sign |
| | | | -INF | -INF | +INF | +INF | if x and y have opposite signs |
| INF | 0 | 0, F, INF | QNaNIndefinite | QNaNIndefinite | QNaNIndefinite | QNaNIndefinite | Signal invalid exception |
| 0 | INF | 0, F, INF | QNaNIndefinite | QNaNIndefinite | QNaNIndefinite | QNaNIndefinite | Signal invalid exception |
| F | INF | +INF | +INF | QNaNIndefinite | QNaNIndefinite | -INF | if $x*y$ and z have the same sign |
| | | | QNaNIndefinite | -INF | +INF | QNaNIndefinite | if $x*y$ and z have opposite signs |
| F | INF | -INF | -INF | QNaNIndefinite | QNaNIndefinite | +INF | if $x*y$ and z have the same sign |
| | | | QNaNIndefinite | +INF | -INF | QNaNIndefinite | if $x*y$ and z have opposite signs |
| F | INF | 0,F | +INF | +INF | -INF | -INF | if $x * y > 0$ |
| | | | -INF | -INF | +INF | +INF | if $x * y < 0$ |
| 0,F | 0,F | INF | +INF | -INF | +INF | -INF | if $z > 0$ |
| | | | -INF | +INF | -INF | +INF | if $z < 0$ |

| x | y | z | $r=x*y+z$ | $r=x*y-z$ | $r = -x*y+z$ | $r = -x*y-z$ | Comment |
|---|---|---|-----------|-----------|--------------|--------------|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | The sign of the result depends on the sign of the operands and on the rounding mode. The product $x*y$ is +0 or -0, depending on the signs of x and y. The summation/subtraction of the zero representing $(x*y)$ and the zero representing z can lead to one of the four cases shown in Table 2-1. |
| 0 | F | 0 | 0 | 0 | 0 | 0 | |
| F | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | F | z | -z | z | -z | |
| 0 | F | F | z | -z | z | -z | |
| F | 0 | F | z | -z | z | -z | |
| F | F | 0 | $x*y$ | $x*y$ | $-x*y$ | $-x*y$ | Rounded to the destination precision, with bounded exponent |
| F | F | F | $x*y+z$ | $x*y-z$ | $-x*y+z$ | $-x*y-z$ | Rounded to the destination precision, with bounded exponent; however, if the exact values of $x*y$ and z are equal in magnitude with signs resulting in the FMA operation producing 0, the rounding behavior described in Table 2-1. |

If unmasked floating-point exceptions are signaled (invalid operation, denormal operand, overflow, underflow, or inexact result) the result register is left unchanged and a floating-point exception handler is invoked.

2.4 ACCESSING YMM 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 (255:128) 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. See Chapter 2, “Programming Considerations with 128-bit SIMD Instructions” for more details.

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.

2.5 MEMORY ALIGNMENT

Memory alignment requirements on VEX-encoded instruction differs from non-VEX-encoded instructions. Memory alignment applies to non-VEX-encoded SIMD instructions in three categories:

- Explicitly-aligned SIMD load and store instructions accessing 16 bytes of memory (e.g. MOVAPD, MOVAPS, MOVDQA, etc.). These instructions always require memory address to be aligned on 16-byte boundary.
- Explicitly-unaligned SIMD load and store instructions accessing 16 bytes or less of data from memory (e.g. MOVUPD, MOVUPS, MOVDQU, MOVQ, MOVD, etc.). These instructions do not require memory address to be aligned on 16-byte boundary.
- The vast majority of arithmetic and data processing instructions in legacy SSE instructions (non-VEX-encoded SIMD instructions) support memory access semantics. When these instructions access 16 bytes of data from memory, the memory address must be aligned on 16-byte boundary.

Most arithmetic and data processing instructions encoded using the VEX prefix and performing memory accesses have more flexible memory alignment requirements than instructions that are encoded without the VEX prefix. Specifically,

- With the exception of explicitly aligned 16 or 32 byte SIMD load/store instructions, most VEX-encoded, arithmetic and data processing instructions operate in a flexible environment regarding memory address alignment, i.e. VEX-encoded instruction with 32-byte or 16-byte load semantics will support unaligned load operation by default. Memory arguments for most instructions with VEX prefix operate normally without causing #GP(0) on any byte-granularity alignment (unlike Legacy SSE instructions). The instructions that require explicit memory alignment requirements are listed in Table 2-4.

Software may see performance penalties when unaligned accesses cross cacheline 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 list of 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 and FMA will generate an #AC(0) fault on misaligned 4 or 8-byte memory references in Ring-3 when CRO.AM=1. 16 and 32-byte memory references will not generate #AC(0) fault. See Table 2-3 for details.

Certain AVX instructions always require 16- or 32-byte alignment (see the complete list of such instructions in Table 2-4). These instructions will #GP(0) if not aligned to 16-byte boundaries (for 16-byte granularity loads and stores) or 32-byte boundaries (for 32-byte loads and stores).

Table 2-3. Alignment Faulting Conditions when Memory Access is Not Aligned

| | | EFLAGS.AC==1 && Ring-3 && CRO.AM == 1 | 0 | 1 |
|----------------------------------|------------------|--|----------|----------|
| Instruction Type | AVX, FMA, | 16- or 32-byte “explicitly unaligned” loads and stores (see Table 2-5) | no fault | no fault |
| | | VEX op YMM, m256 | no fault | no fault |
| | | VEX op XMM, m128 | no fault | no fault |
| | | “explicitly aligned” loads and stores (see Table 2-4) | #GP(0) | #GP(0) |
| | | 2, 4, or 8-byte loads and stores | no fault | #AC(0) |
| | SSE | 16 byte “explicitly unaligned” loads and stores (see Table 2-5) | no fault | no fault |
| | | op XMM, m128 | #GP(0) | #GP(0) |
| | | “explicitly aligned” loads and stores (see Table 2-4) | #GP(0) | #GP(0) |
| 2, 4, or 8-byte loads and stores | | no fault | #AC(0) | |

Table 2-4. instructions Requiring Explicitly Aligned Memory

| Require 16-byte alignment | Require 32-byte alignment |
|----------------------------------|----------------------------------|
| (V)MOVDQA xmm, m128 | VMOVDQA ymm, m256 |
| (V)MOVDQA m128, xmm | VMOVDQA m256, ymm |
| (V)MOVAPS xmm, m128 | VMOVAPS ymm, m256 |

Table 2-4. instructions Requiring Explicitly Aligned Memory

| Require 16-byte alignment | Require 32-byte alignment |
|---------------------------|---------------------------|
| (V)MOVAPS m128, xmm | VMOVAPS m256, ymm |
| (V)MOVAPD xmm, m128 | VMOVAPD ymm, m256 |
| (V)MOVAPD m128, xmm | VMOVAPD m256, ymm |
| (V)MOVNTPS m128, xmm | VMOVNTPS m256, ymm |
| (V)MOVNTPD m128, xmm | VMOVNTPD m256, ymm |
| (V)MOVNTDQ m128, xmm | VMOVNTDQ m256, ymm |
| (V)MOVNTDQA xmm, m128 | |

Table 2-5. Instructions Not Requiring Explicit Memory Alignment

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

2.6 SIMD FLOATING-POINT EXCEPTIONS

AVX and FMA instructions can generate SIMD floating-point exceptions (#XM) and respond to exception masks in the same way as Legacy SSE instructions. When CR4.OSXMMEXCPT=0 any unmasked FP exceptions generate an Undefined Opcode exception (#UD).

AVX FP exceptions are created in a similar fashion (differing only in number of elements) to Legacy SSE and SSE2 instructions capable of generating SIMD floating-point exceptions.

AVX introduces no new arithmetic operations (AVX floating-point are analogues of existing Legacy SSE instructions). FMA introduces new arithmetic operations, detailed FMA numeric behavior are described in Section 2.3.

2.7 INSTRUCTION EXCEPTION SPECIFICATION

To use this reference of instruction exceptions, look at each instruction for a description of the particular exception type of interest. For example, ADDPS contains the entry:

“See Exceptions Type 2”

In this entry, *“Type2”* can be looked up in Table 2-6.

The instruction’s corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

Table 2-6. Exception class description

| Exception Class | Instruction set | Mem arg | Floating-Point Exceptions (#XM) |
|-----------------|------------------------|--------------------------------------|---------------------------------|
| Type 1 | AVX, Legacy SSE | 16/32 byte explicitly aligned | none |
| Type 2 | AVX, FMA Legacy SSE | 16/32 byte not explicitly aligned | yes |
| Type 3 | AVX, FMA Legacy SSE | < 16 byte | yes |
| Type 4 | AVX, Legacy SSE | 16/32 byte not explicitly aligned | no |
| Type 5 | AVX, Legacy SSE | < 16 byte | no |
| Type 6 | AVX (no Legacy SSE) | Varies | (At present, none do) |
| Type 7 | AVX, Legacy SSE | none | none |
| Type 8 | AVX | none | none |
| Type 9 | AVX | 4 byte | none |

See Table 2-7 for lists of instructions in each exception class.

Table 2-7. Instructions in each Exception Class

| Exception Class | Instruction |
|-----------------|---|
| Type 1 | (V)MOVAPD, (V)MOVAPS, (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS |
| Type 2 | (V)ADDPD, (V)ADDPS, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, (V)FMADDPD, (V)FMADDPS, (V)FMADDSUBPD, (V)FMADDSUBPS, (V)FMSUBADDPD, (V)FMSUBADDPS, (V)FMSUBPD, (V)FMSUBPS, (V)FNMADDPD, (V)FNMADDPS, (V)FNMSUBPD, (V)FNMSUBPS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS |
| Type 3 | (V)ADDS, (V)ADDS, (V)CMPD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTSS2SI, (V)CVTSS2SI, (V)DIVSD, (V)DIVSS, (V)FMADDS, (V)FMADDS, (V)FMSUBSD, (V)FMSUBSS, (V)FNMADDS, (V)FNMADDS, (V)FNMSUBSD, (V)FNMSUBSS, (V)MAXSD, (V)MAXSS, (V)MINS, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS |
| Type 4 | (V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, (V)BLENDVPD, (V)BLENDVPS, (V)LDDQU, (V)MASKMOVDQU, (V)PTEST, (V)TESTPS, (V)TESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABS, (V)PABS, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, |

| Exception Class | Instruction |
|-----------------|---|
| | (V)PMADDWD, (V)PMADDUBSW, (V)PMAXSB, (V)PMAXSW, (V)PMAXSD, (V)PMAXUB, (V)PMAXUW, (V)PMAXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFHW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS |
| Type 5 | (V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX |
| Type 6 | VEXTRACTF128, VPERMILPD, VPERMIL2PD, VPERMILPS, VPERMIL2PS, VPERM2F128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD |
| Type 7 | (V)MOVLHPS, (V)MOVHLP, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMSKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ |
| Type 8 | VZEROALL, VZEROUPPER |
| Type 9 | VLDMXCSR*, VSTMXCSR |

(*) - Additional exception restrictions are present - see the Instruction description for details

Table 2-7 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-9 through Table 2-15, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-8 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.L field.

Table 2-8. #UD Exception and VEX.L Field Encoding

| Exception Class | #UD If VEX.L = 0 | #UD If VEX.L = 1 |
|-----------------|--|---|
| Type 1 | | VMOVNTDQA |
| Type 2 | | VDPPD |
| Type 3 | | |
| Type 4 | | VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMASB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRSW, VPMULHW/LW, VPMULLD, VPMULUDQ, VPMULDQ, VPOR, VPSADBW, VPSHUFB/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/WD/DQ, VPUNPCKHQDQ, VPUNPCKLBW/WD/DQ, VPUNPCKLQDQ, VPXOR |
| Type 5 | | VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX |
| Type 6 | VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128, | |
| Type 7 | | VMOVLHPS, VMOVHLPs, VPMOVMskB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ |
| Type 8 | | |
| Type 9 | | VLDMXCSR, VSTMXCSR |

2.7.1 Exceptions Type 1 (Aligned memory reference)

Table 2-9. Type 1 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|----------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | | | X | X | VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | X | X | X | X | Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. |
| | 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 | 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 | VEX.256: Memory operand is not 32-byte aligned VEX.128: Memory operand is not 16-byte aligned |
| | X | X | X | X | Legacy SSE: 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 |

2.7.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

Table 2-10. Type 2 Class Exception Conditions

| Exception | Real | Virtual 8086 | Protected and Compatibility | 64-bit | Cause of Exception |
|------------------------------------|------|--------------|-----------------------------|--------|---|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | X | X | X | X | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | | | X | X | VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | X | X | X | X | Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. |
| | 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 | 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 | X | X | Legacy SSE: 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 |
| SIMD Floating-point Exception, #XM | X | X | X | X | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1 |

2.7.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-11. Type 3 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|------------------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | X | X | X | X | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | | | X | X | VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | X | X | X | X | Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. |
| | 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 | 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 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 and CR4.OSXMMEXCPT[bit 10] = 1 |

2.7.4 Exceptions Type 4 (>=16 Byte mem arg no alignment, no floating-point exceptions)

Table 2-12. Type 4 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|----------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | | | X | X | VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | X | X | X | X | Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. |
| | 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 | 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 | X | X | Legacy SSE: 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 |

2.7.5 Exceptions Type 5 (<16 Byte mem arg and no FP exceptions)

Table 2-13. Type 5 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|----------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | | | X | X | VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | X | X | X | X | Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. |
| | 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 | 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 is made while the current privilege level is 3. |

2.7.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

Table 2-14. Type 6 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|----------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | | | X | X | If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | | | 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 | 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 is made while the current privilege level is 3. |

2.7.7 Exceptions Type 7 (No FP exceptions, no memory arg)

Table 2-15. Type 7 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|---------------------------|------|---------------|-----------------------------|--------|---|
| Invalid Opcode, #UD | X | X | | | VEX prefix |
| | | | X | X | VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. |
| | X | X | X | X | Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. |
| | 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 | X | If any corresponding CPUID feature flag is '0' |
| Device Not Available, #NM | | | X | X | If CR0.TS[bit 3]=1 |

2.7.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-16. Type 8 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|---------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD | X | X | | | Always in Real or Virtual 80x86 mode |
| | | | X | X | If XFEATURE_ENABLED_MASK[2:1] != '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv != 1111B. |
| | X | X | X | X | If proceeded by a LOCK prefix (F0H) |
| Device Not Available, #NM | | | X | X | If CR0.TS[bit 3]=1. |

2.7.9 Exception Type 9 (AVX)

Table 2-17. Type 9 Class Exception Conditions

| Exception | Real | Virtual 80x86 | Protected and Compatibility | 64-bit | Cause of Exception |
|----------------------------|------|---------------|-----------------------------|--------|--|
| Invalid Opcode, #UD | X | X | | | Always in Real or Virtual 80x86 mode |
| | | | X | X | If CR4.OSXSAVE[bit 18]=0. If CR0.EM[bit 2] = 1. If CPUID.01H.ECX.AVX[bit 28]=0 If VEX.L = 1 |
| | | | X | X | If preceded by a LOCK prefix (FOH) |
| | | | X | X | If any REX, F2, F3, or 66 prefixes precede a VEX prefix |
| 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 | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

2.8 PROGRAMMING CONSIDERATIONS WITH 128-BIT SIMD INSTRUCTIONS

VEX-encoded SIMD instructions generally operate on the 256-bit YMM register state. In contrast, non-VEX encoded instructions (e.g from SSE to AES) operating on XMM registers only access the lower 128-bit of YMM registers. Processors supporting both 256-bit VEX-encoded instruction and legacy 128-bit SIMD instructions has internal state to manage the upper and lower halves of the YMM states. Functionally, VEX-encoded SIMD instructions can be intermixed with legacy SSE instructions (non-VEX-

encoded SIMD instructions operating on XMM registers). However, there is a performance impact with intermixing VEX-encoded SIMD instructions (AVX, FMA) and Legacy SSE instructions that only operate on the XMM register state.

The general programming considerations to realize optimal performance are the following:

- Minimize transition delays and partial register stalls with YMM registers accesses: Intermixed 256-bit, 128-bit or scalar SIMD instructions that are encoded with VEX prefixes have no transition delay due to internal state management. Sequences of legacy SSE instructions (including SSE2, and subsequent generations non-VEX-encoded SIMD extensions) that are not intermixed with VEX-encoded SIMD instructions are not subject to transition delays.
- When an application must employ AVX and/or FMA, along with legacy SSE code, it should minimize the number of transitions between VEX-encoded instructions and legacy, non-VEX-encoded SSE code. Section 2.8.1 provides recommendation for software to minimize the impact of transitions between VEX-encoded code and legacy SSE code.

2.8.1 Clearing Upper YMM State Between AVX and Legacy SSE Instructions

There is no transition penalty if an application clears the upper bits of all YMM registers (set to '0') via `VZERoupper`, `VZEROALL`, before transitioning between AVX instructions and legacy SSE instructions. Note: clearing the upper state via sequences of `XORPS` or loading '0' values individually may be useful for breaking dependency, but will not avoid state transition penalties.

Example 1: an application using 256-bit AVX instructions makes calls to a library written using Legacy SSE instructions. This would encounter a delay upon executing the first Legacy SSE instruction in that library and then (after exiting the library) upon executing the first AVX instruction. To eliminate both of these delays, the user should execute the instruction `VZERoupper` prior to entering the legacy library and (after exiting the library) before executing in a 256-bit AVX code path.

Example 2: a library using 256-bit AVX instructions is intended to support other applications that uses legacy SSE instructions. Such a library function should execute `VZERoupper` prior to executing other VEX-encoded instructions. The library function should issue `VZERoupper` at the end of the function before it returns to the calling application. This will prevent the calling application to experience delay when it starts to execute legacy SSE code.

2.8.2 Using AVX 128-bit Instructions Instead of Legacy SSE instructions

Applications using AVX and FMA should migrate legacy 128-bit SIMD instructions to their 128-bit AVX equivalents. AVX supplies the full complement of 128-bit SIMD instructions except for AES and PCLMULQDQ.

2.8.3 Unaligned Memory Access and Buffer Size Management

The majority of AVX instructions support loading 16/32 bytes from memory without alignment restrictions (A number non-VEX-encoded SIMD instructions also don't require 16-byte address alignment, e.g. MOVDQU, MOVUPS, MOVUPD, LDDQU, PCMPSTRI/PCMPSTRM/PCMPISTRI/PCMPISTRM). A buffer size management issue related to unaligned SIMD memory access is discussed here.

The size requirements for memory buffer allocation should consider unaligned SIMD memory semantics and application usage. Frequently a caller function may pass an address pointer in conjunction with a length parameter. From the caller perspective, the length parameter usually corresponds to the limit of the allocated memory buffer range, or it may correspond to certain application-specific configuration parameter that have indirect relationship with valid buffer size.

For certain types of application usage, it may be desirable to make distinctions between valid buffer range limit versus other application specific parameters related memory access patterns, examples of the latter may be stride distance, frame dimensions, etc. There may be situations that a callee wishes to load 16-bytes of data with parts of the 16-bytes lying outside the valid memory buffer region to take advantage of the efficiency of SIMD load bandwidth and discard invalid data elements outside the buffer boundary. An example of this may be in video processing of frames having dimensions that are not modular 16 bytes.

To support the added margin of safety in situations of buffer size allocation and iterative pointer advancement occurring across modules of different software visibility. The standard programming practice of caller function allocation of buffer size based on non-SIMD processing requirement should consider an added padding size to support newer SIMD extensions offering more lax alignment restrictions. The extra padding space can prevent the rare occurrence of access rights violation described below:

- A present page in the linear address space being used by ring 3 code is followed by a page owned by ring 0 code,
- A caller routine allocates a memory buffer without adding extra pad space and passes the buffer address to a callee routine,
- A callee routine implements an iterative processing algorithm by advancing an address pointer relative to the buffer address using SIMD instructions with unaligned 16/32 load semantics
- The callee routine may choose to load 16/32 bytes near buffer boundary with the intent to discard invalid data outside the data buffer allocated by the caller.

APPLICATION PROGRAMMING MODEL

- If the valid data buffer extends to the end of the present page, unaligned 16/32 byte loads near the end of a present page may spill over to the subsequent ring-0 page and causing a #GP.

As a general rule, the minimal padding size should be the width the SIMD register that might be used in conjunction with unaligned SIMD memory access.

2.9 CPUID INSTRUCTION

CPUID—CPU Identification

| Opcode | Instruction | 64-Bit Mode | Compat/ 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.¹ The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

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

Table 2-18 shows information returned, depending on the initial value loaded into the EAX register. Table 2-19 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,

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

and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

“Serializing Instructions” in Chapter 7, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A AP-485, Intel Processor Identification and the CPUID Instruction* (Order Number 241618)

Table 2-18. Information Returned by CPUID Instruction

| Initial EAX Value | Information Provided about the Processor | |
|-------------------|--|--|
| 0H | <i>Basic CPUID Information</i> | |
| | EAX EBX ECX EDX | Maximum Input Value for Basic CPUID Information (see Table 2-19) “Genu” “ntel” “intel” |
| 01H | EAX EBX ECX EDX | Version Information: Type, Family, Model, and Stepping ID (see Figure 2-2) Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value * 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID Feature Information (see Figure 2-3 and Table 2-21) Feature Information (see Figure 2-4 and Table 2-22) NOTES: * 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 EBX ECX EDX | Cache and TLB Information (see Table 2-23) Cache and TLB Information Cache and TLB Information Cache and TLB Information |
| 03H | EAX EBX ECX EDX | Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor |
|-------------------|--|
| | <p>NOTES:</p> <p>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.</p> |
| | <p>See AP-485, <i>Intel Processor Identification and the CPUID Instruction</i> (Order Number 241618) for more information on PSN.</p> |
| | <p>CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).</p> |
| | <p><i>Deterministic Cache Parameters Leaf</i></p> |
| <p>04H</p> | <p>NOTES:</p> <p>Leaf 04H output depends on the initial value in ECX.</p> <p>See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 1-58.</p> <p>Bits 4-0: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved</p> <p>Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache</p> <p>Bits 13-10: Reserved Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache*, ** Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package*, ***, ****</p> <p>EBX Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity*</p> <p>ECX Bits 31-00: S = Number of Sets*</p> |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor | |
|---------------------------|--|---|
| | EDX | Bit 0: WBINVD/INVD behavior on lower level caches Bit 10: Write-Back Invalidate/Invalidate 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 1: Cache Inclusiveness 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bits 31-02: Reserved = 0 |
| | NOTES: * Add one to the return value to get the result. **The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache *** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ****The returned value is constant for valid initial values in ECX. Valid ECX values start from 0. | |
| <i>MONITOR/MWAIT Leaf</i> | | |
| 5H | EAX | Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0 Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0 Bits 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bits 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31 - 02: Reserved |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor | |
|-------------------|--|--|
| | EDX | Bits 03 - 00: Number of C0* sub C-states supported using MWait Bits 07 - 04: Number of C1* sub C-states supported using MWait Bits 11 - 08: Number of C2* sub C-states supported using MWait Bits 15 - 12: Number of C3* sub C-states supported using MWait Bits 19 - 16: Number of C4* sub C-states supported using MWait Bits 31 - 20: Reserved = 0 NOTE: * 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> | |
| 6H | EAX | Bits 00: Digital temperature sensor is supported if set Bits 01: Intel Dynamic Acceleration Enabled Bits 31 - 02: Reserved |
| | EBX | Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor 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 Bits 31 - 01: Reserved = 0 |
| | EDX | Reserved = 0 |
| | <i>Direct Cache Access Information Leaf</i> | |
| 09H | EAX | Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H) |
| | EBX | Reserved |
| | ECX | Reserved |
| | EDX | Reserved |
| | <i>Architectural Performance Monitoring Leaf</i> | |
| 0AH | EAX | Bits 07 - 00: Version ID of architectural performance monitoring Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor Bits 23 - 16: Bit width of general-purpose, performance monitoring counter Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor | |
|---|--|--|
| | EBX | Bit 0: Core cycle event not available if 1 Bit 1: Instruction retired event not available if 1 Bit 2: Reference cycles event not available if 1 Bit 3: Last-level cache reference event not available if 1 Bit 4: Last-level cache misses event not available if 1 Bit 5: Branch instruction retired event not available if 1 Bit 6: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0 |
| | ECX EDX | Reserved = 0 Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1) Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1) Reserved = 0 |
| <i>Extended Topology Enumeration Leaf</i> | | |
| OBH | EAX | NOTES: Most of Leaf OBH output depends on the initial value in ECX. EDX output do not vary with initial value in ECX. ECX[7:0] output always reflect initial value in ECX. All other output value for an invalid initial value in ECX are 0 This leaf exists if EBX[15:0] contain a non-zero value. Bits 4-0: 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. Bits 31-5: Reserved. |
| OBH | EBX | Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved. |
| | ECX EDX | Bits 07 - 00: Level number. Same value in ECX input Bits 15 - 08: Level type***. Bits 31 - 16: Reserved. Bits 31- 0: x2APIC ID the current logical processor. |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | <p style="text-align: center;">Information Provided about the Processor</p> <p>NOTES: * 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: 0: invalid 1: SMT 2: Core 3-255: Reserved</p> |
| | <i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i> |
| 0DH | <p>NOTES: Leaf 0DH main leaf (ECX = 0).</p> <p>EAX Bits 31-0: 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 XFEATURE_ENABLED_MASK is reserved.</p> <p>EBX Bits 31-0: Maximum size (bytes) required by enabled features in XFEATURE_ENABLED_MASK. May be different than ECX when features at the end of the save area are not enabled.</p> <p>ECX Bit 31-0: Maximum size (bytes) of the XSAVE/XRSTOR save area required by all HW supported features, i.e all the valid bit fields in XFEATURE_ENABLED_MASK. This includes the size needed for the XSAVE.HEADER.</p> |
| | <p>EDX Bit 31-0: Reports the valid bit fields of the upper 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XFEATURE_ENABLED_MASK is reserved</p> |
| | <i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i> |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor | |
|-------------------|---|---|
| | EAX | Reserved |
| | EBX | Reserved |
| | ECX | Reserved |
| | EDX | Reserved |
| | <i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)</i> | |
| 0DH | <p>NOTES:</p> <p>Leaf 0DH output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.</p> | |
| | EAX | Bits 31-0: The size in bytes of the save area for an extended state associated with a valid sub-leaf index, <i>n</i> . Each valid sub-leaf index maps to a valid bit in XFEATURE_ENABLED_MASK starting at bit position 2. This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*. |
| | EBX | Bits 31-0: The offset in bytes of the save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*. |
| | ECX | This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*; otherwise it is reserved. |
| | EDX | This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*; otherwise it is reserved. |
| | *The highest valid sub-leaf index, <i>n</i> , is (POPCNT(CPUID.(EAX=0D, ECX=0):EAX) + POPCNT(CPUID.(EAX=0D, ECX=0):EDX) - 1) | |
| | <i>Extended Function CPUID Information</i> | |
| 8000000H | EAX | Maximum Input Value for Extended Function CPUID Information (see Table 2-19). |
| | EBX | Reserved |
| | ECX | Reserved |
| | EDX | Reserved |
| 8000001H | EAX | Extended Processor Signature and Feature Bits. |
| | EBX | Reserved |
| | ECX | Bit 0: LAHF/SAHF available in 64-bit mode Bits 31-1 Reserved |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor | |
|-------------------|--|---|
| | EDX | Bits 10-0: Reserved Bit 11: SYSCALL/SYSRET available (when in 64-bit mode) Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 28-21: Reserved = 0 Bit 29: Intel [®] 64 Architecture available if 1 Bits 31-30: Reserved = 0 |
| 80000002H | EAX EBX ECX EDX | Processor Brand String Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued |
| 80000003H | EAX EBX ECX EDX | Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued |
| 80000004H | EAX EBX ECX EDX | Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued |
| 80000005H | EAX EBX ECX EDX | Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0 |
| 80000006H | EAX EBX ECX EDX | Reserved = 0 Reserved = 0 Bits 7-0: Cache Line size in bytes Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units Reserved = 0 NOTES: * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative |

Table 2-18. Information Returned by CPUID Instruction (Contd.)

| Initial EAX Value | Information Provided about the Processor | |
|-------------------|--|---|
| 80000007H | EAX EBX ECX EDX | Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0 |
| 80000008H | EAX | Virtual/Physical Address size Bits 7-0: #Physical Address Bits* Bits 15-8: #Virtual Address Bits Bits 31-16: Reserved = 0 |
| | EBX ECX EDX | Reserved = 0 Reserved = 0 Reserved = 0 NOTES: * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field. |

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 2-19) 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 0, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 2-19) and is processor specific.

Table 2-19. 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 |
| 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 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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

When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 2-2). 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 2-20 for available processor type values. Stepping IDs are provided as needed.

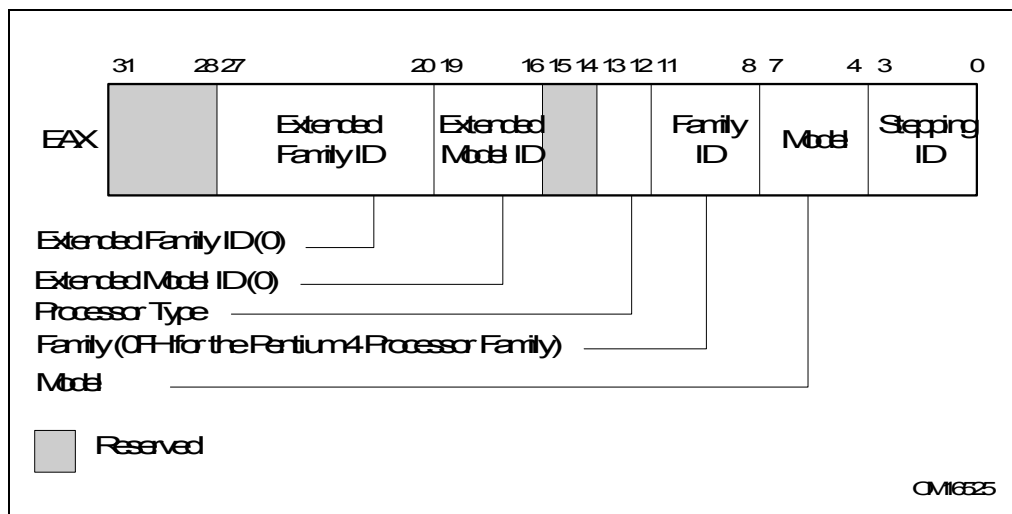


Figure 2-2. Version Information Returned by CPUID in EAX

Table 2-20. 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 AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) and Chapter 14 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 = 1: Returns Additional Information in EBX

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

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

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

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

- Figure 2-3 and Table 2-21 show encodings for ECX.
- Figure 2-4 and Table 2-22 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.

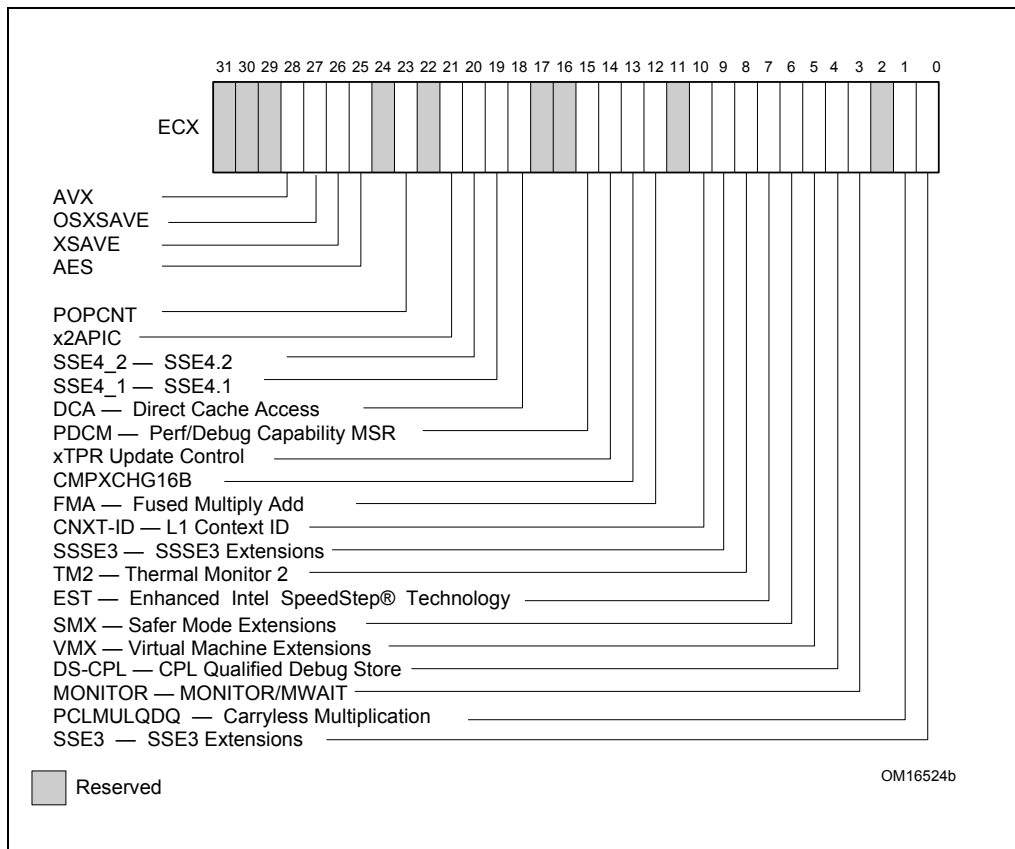


Figure 2-3. Feature Information Returned in the ECX Register

Table 2-21. Feature Information Returned in the ECX Register

| Bit # | Mnemonic | Description |
|-------|-----------|---|
| 0 | SSE3 | Streaming SIMD Extensions 3 (SSE3). A value of 1 indicates the processor supports this technology. |
| 1 | PCLMULQDQ | A value of 1 indicates the processor supports PCLMULQDQ instruction |
| 2 | Reserved | Reserved |

Table 2-21. Feature Information Returned in the ECX Register (Contd.)

| Bit # | Mnemonic | Description |
|-------|---------------------|--|
| 3 | MONITOR | MONITOR/MWAIT. A value of 1 indicates the processor supports this feature. |
| 4 | DS-CPL | CPL Qualified Debug Store. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL. |
| 5 | VMX | Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology |
| 6 | SMX | Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”. |
| 7 | EST | Enhanced Intel SpeedStep® technology. A value of 1 indicates that the processor supports this technology. |
| 8 | TM2 | Thermal Monitor 2. A value of 1 indicates whether the processor supports this technology. |
| 9 | SSSE3 | A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor |
| 10 | CNXT-ID | L1 Context ID. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details. |
| 11 | Reserved | Reserved |
| 12 | FMA | A value of 1 indicates the processor supports FMA extensions using YMM state. |
| 13 | CMPXCHG16B | CMPXCHG16B Available. A value of 1 indicates that the feature is available. See the “CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes” section in this chapter for a description. |
| 14 | xTPR Update Control | xTPR Update Control. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23]. |
| 15 | PDCM | Perfmon and Debug Capability: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES. |
| 16-17 | Reserved | Reserved |
| 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. |

Table 2-21. Feature Information Returned in the ECX Register (Contd.)

| Bit # | Mnemonic | Description |
|---------|----------|--|
| 22 | Reserved | Reserved |
| 23 | POPCNT | A value of 1 indicates that the processor supports the POPCNT instruction. |
| 24 | Reserved | Reserved |
| 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. |
| 31 - 29 | Reserved | Reserved |

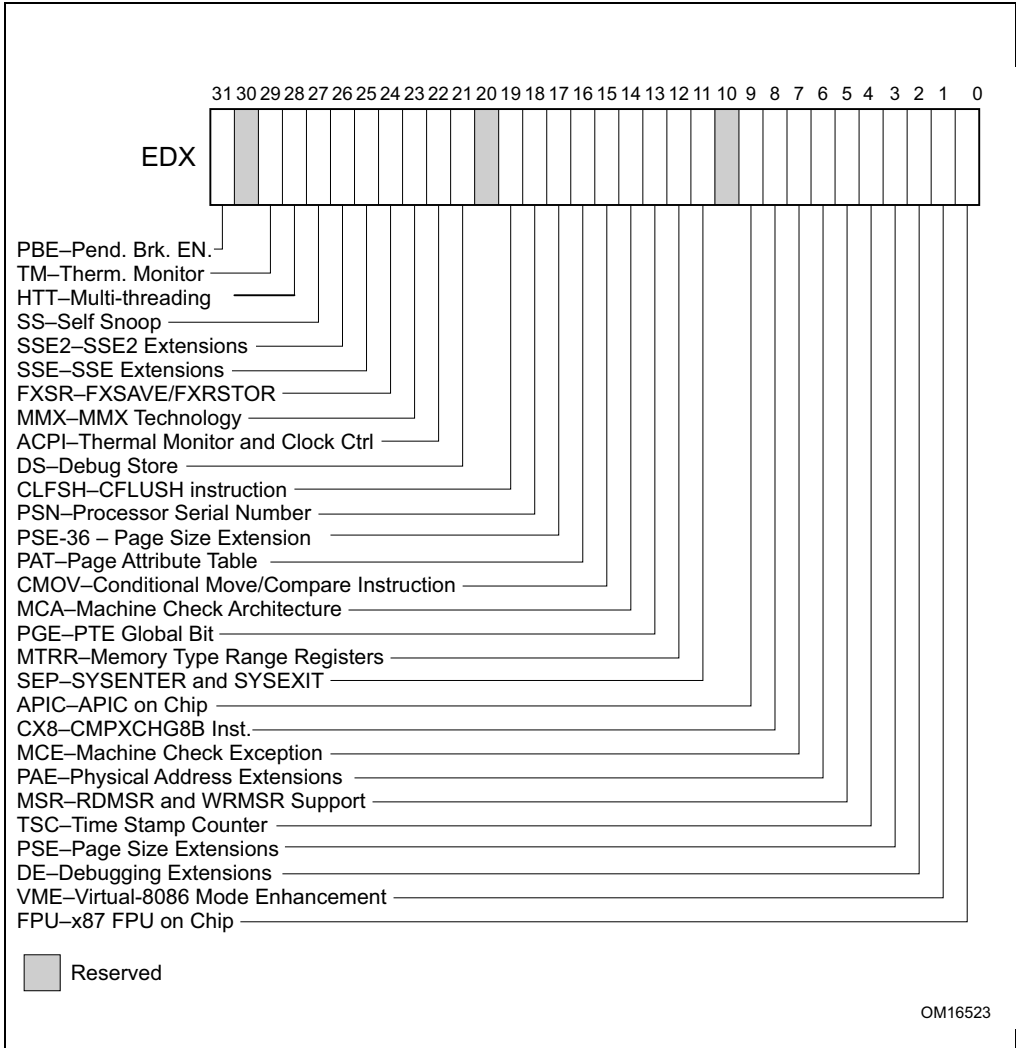


Figure 2-4. Feature Information Returned in the EDX Register

Table 2-22. More on Feature Information Returned in the EDX Register

| Bit # | Mnemonic | Description |
|-------|----------|--|
| 0 | FPU | floating-point Unit On-Chip. The processor contains an x87 FPU. |
| 1 | VME | Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags. |
| 2 | DE | Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5. |
| 3 | PSE | Page Size Extension. Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs. |
| 4 | TSC | Time Stamp Counter. The RDTSC instruction is supported, including CR4.TSD for controlling privilege. |
| 5 | MSR | Model Specific Registers RDMSR and WRMSR Instructions. The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent. |
| 6 | PAE | Physical Address Extension. Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific. |
| 7 | MCE | Machine Check Exception. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature. |
| 8 | CX8 | CMPXCHG8B Instruction. The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic). |
| 9 | APIC | APIC On-Chip. The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated). |
| 10 | Reserved | Reserved |
| 11 | SEP | SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT and associated MSRs are supported. |
| 12 | MTRR | Memory Type Range Registers. MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported. |

Table 2-22. More on Feature Information Returned in the EDX Register (Contd.)

| Bit # | Mnemonic | Description |
|-------|----------|---|
| 13 | PGE | PTE Global Bit. 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 | Machine Check Architecture. The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported. |
| 15 | CMOV | Conditional Move Instructions. The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported |
| 16 | PAT | Page Attribute Table. Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address. |
| 17 | PSE-36 | 36-Bit Page Size Extension. 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 | Processor Serial Number. The processor supports the 96-bit processor identification number feature and the feature is enabled. |
| 19 | CLFSH | CLFLUSH Instruction. CLFLUSH Instruction is supported. |
| 20 | Reserved | Reserved |
| 21 | DS | Debug Store. The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 18, “Debugging and Performance Monitoring,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B</i>). |
| 22 | ACPI | Thermal Monitor and Software Controlled Clock Facilities. The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control. |
| 23 | MMX | Intel MMX Technology. The processor supports the Intel MMX technology. |
| 24 | FXSR | FXSAVE and FXRSTOR Instructions. The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating-point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions. |

Table 2-22. More on Feature Information Returned in the EDX Register (Contd.)

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

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

When CPUID executes with EAX set to 2, 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 2 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 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 2-23 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 2-23. 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 |

Table 2-23. Encoding of Cache and TLB Descriptors (Contd.)

| Descriptor Value | Cache or TLB Description |
|------------------|--|
| 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 |
| 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); 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 |

Table 2-23. Encoding of Cache and TLB Descriptors (Contd.)

| Descriptor Value | Cache or TLB Description |
|------------------|--|
| 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- μ op, 8-way set associative |
| 71H | Trace cache: 16 K- μ op, 8-way set associative |
| 72H | Trace cache: 32 K- μ 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 |
| 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 |

Table 2-23. Encoding of Cache and TLB Descriptors (Contd.)

| Descriptor Value | Cache or TLB Description |
|------------------|---|
| 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- μ 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 = 4: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 4 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 2-18.

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=4 and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 7, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 5: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 5, 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 2-18.

INPUT EAX = 6: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 6, the processor returns information about thermal and power management features. See Table 2-18.

INPUT EAX = 9: Returns Direct Cache Access Information

When CPUID executes with EAX set to 9, the processor returns information about Direct Cache Access capabilities. See Table 2-18.

INPUT EAX = 10: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 10, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 2-18) is greater than Pn 0. See Table 2-18.

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 18, “Debugging and Performance Monitoring,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

INPUT EAX = 11: Returns Extended Topology Information

When CPUID executes with EAX set to 11, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is $\geq 0BH$, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value.

INPUT EAX = 13: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 13 and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 2-18.

When CPUID executes with EAX set to 13 and ECX = n ($n > 1$ and less than the number of non-zero bits in CPUID. (EAX=0DH, ECX= 0H).EAX and 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 2-18.

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 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 2-5 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.

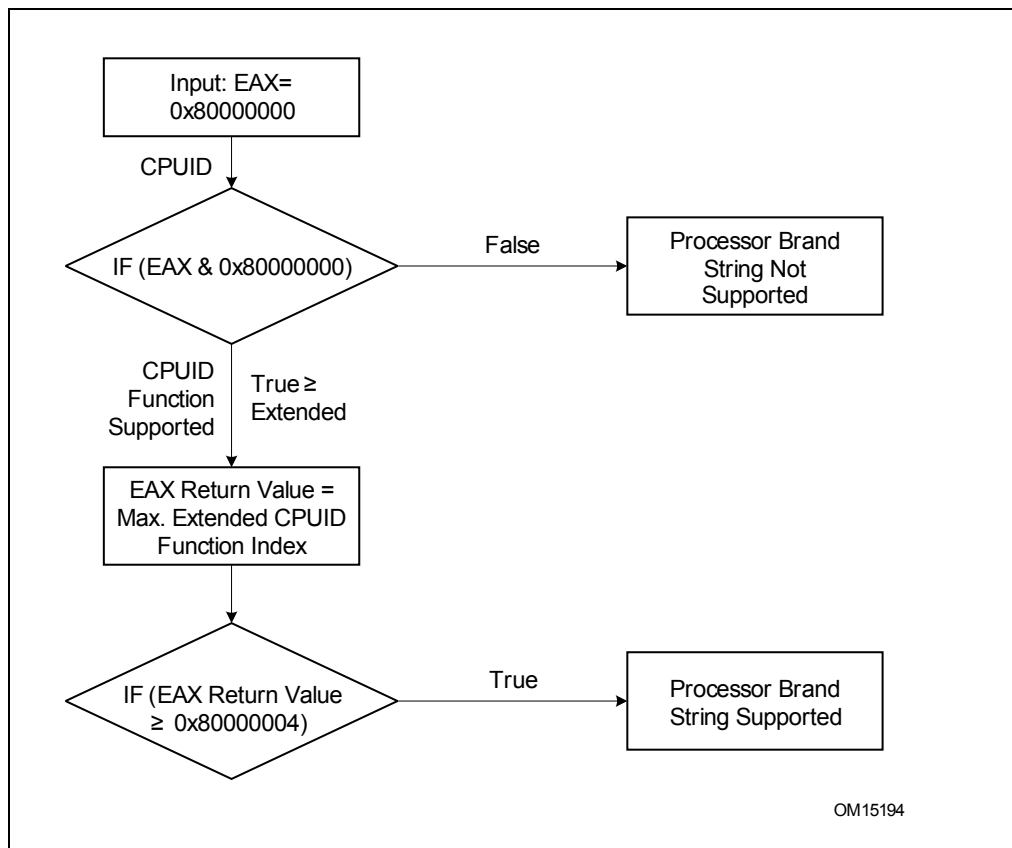


Figure 2-5. Determination of Support for the Processor Brand String

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 2-24 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 2-24. Processor Brand String Returned with Pentium 4 Processor

| EAX Input Value | Return Values | ASCII Equivalent |
|-----------------|---------------|------------------|
|-----------------|---------------|------------------|

Table 2-24. Processor Brand String Returned with Pentium 4 Processor (Contd.)

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

Extracting the Maximum Processor Frequency from Brand Strings

Figure 2-6 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.

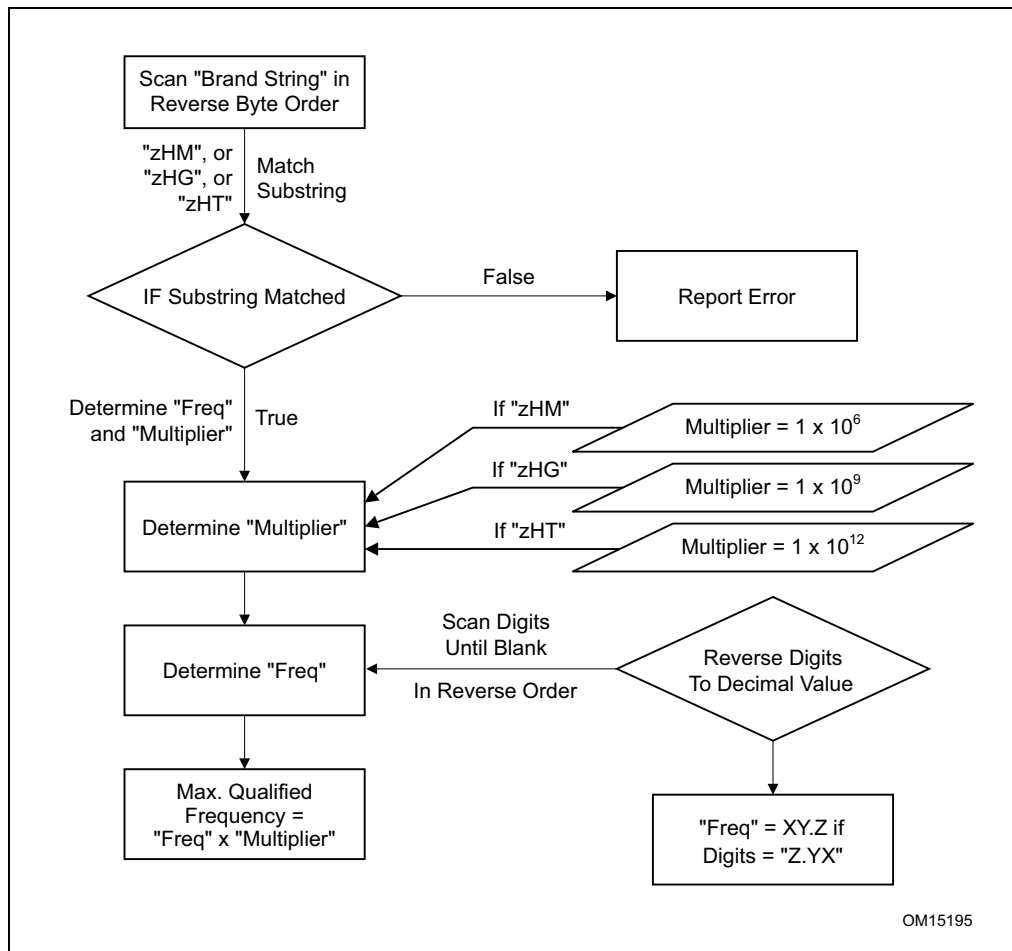


Figure 2-6. Algorithm for Extracting Maximum Processor Frequency

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 associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that

do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 2-25 shows brand indices that have identification strings associated with them.

Table 2-25. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

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

NOTES:

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

IA-32 Architecture Compatibility

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

Operation

IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF

EAX = 0:

EAX ← Highest basic function input value understood by CPUID;

EBX ← Vendor identification string;

EDX ← Vendor identification string;

ECX ← Vendor identification string;

BREAK;

EAX = 1H:

EAX[3:0] ← Stepping ID;

EAX[7:4] ← Model;

EAX[11:8] ← Family;

EAX[13:12] ← Processor type;

EAX[15:14] ← Reserved;

EAX[19:16] ← Extended Model;

EAX[27:20] ← Extended Family;

EAX[31:28] ← Reserved;

EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)

EBX[15:8] ← CLFLUSH Line Size;

EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)

EBX[24:31] ← Initial APIC ID;

ECX ← Feature flags; (* See Figure 2-3. *)

EDX ← Feature flags; (* See Figure 2-4. *)

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 2-18. *)

EBX ← Deterministic Cache Parameters Leaf;

ECX ← Deterministic Cache Parameters Leaf;

EDX ← Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:

EAX ← MONITOR/MWAIT Leaf; (* See Table 2-18. *)

EBX ← MONITOR/MWAIT Leaf;

ECX ← MONITOR/MWAIT Leaf;

EDX ← MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:

EAX ← Thermal and Power Management Leaf; (* See Table 2-18. *)

EBX ← Thermal and Power Management Leaf;

ECX ← Thermal and Power Management Leaf;

EDX ← Thermal and Power Management Leaf;

BREAK;

EAX = 7H or 8H:

EAX ← Reserved = 0;

EBX ← Reserved = 0;

ECX ← Reserved = 0;

EDX ← Reserved = 0;

BREAK;

EAX = 9H:

EAX ← Direct Cache Access Information Leaf; (* See Table 2-18. *)

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 2-18. *)

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 2-18. *)

EBX ← Extended Topology Enumeration Leaf;

ECX ← Extended Topology Enumeration Leaf;

EDX ← Extended Topology Enumeration Leaf;

BREAK;

APPLICATION PROGRAMMING MODEL

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 2-18. *)

EBX ← Processor Extended State Enumeration Leaf;

ECX ← Processor Extended State Enumeration Leaf;

EDX ← Processor Extended State 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 2-18.*);

EDX ← Extended Feature Bits (* See Table 2-18. *);

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. |
|-----|--|

CHAPTER 3

SYSTEM PROGRAMMING MODEL

This chapter describes the operating system programming considerations for AVX. The AES extension and PCLMULQDQ instruction follow the same system software requirements for XMM state support and SIMD floating-point exception support as SSE2, SSE3, SSSE3, SSE4 (see Chapter 12 of *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A*).

The AVX and FMA extensions operate on 256-bit YMM registers, and require operating system to support processor extended state management using XSAVE/XRSTOR instructions. VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST follow the same system programming requirements as AVX and FMA instructions operating on YMM states.

The basic requirements for an operating system using XSAVE/XRSTOR to manage processor extended states for current and future Intel Architecture processors can be found in Chapter 12 of *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A*. This chapter covers additional requirements for OS to support YMM state.

3.1 YMM STATE, VEX PREFIX AND SUPPORTED OPERATING MODES

AVX and FMA instructions comprises of 256-bit and 128-bit instructions that operates on YMM states via VEX prefix encoding. SIMD instructions operating on XMM states (i.e. not accessing the upper 128 bits of YMM) generally do not use VEX prefix.

For processors that support YMM states, the YMM state exists in all operating modes. However, the available interfaces to access YMM states may vary in different modes. The processor's support for instruction extensions that employ VEX prefix encoding is independent of the processor's support for YMM state.

Instructions requiring VEX 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 255:128 of YMM register state are maintained across transitions into and out of these modes. Because, XSAVE/XRSTOR instruction can operate in all operating modes, it is possible that the processor's YMM register state can be modified by software in any operating mode by executing XRSTOR. The YMM registers can be updated by XRSTOR using the state information stored in the XSAVE/XRSTOR area residing in memory.

3.2 YMM STATE MANAGEMENT

Operating systems must use the XSAVE/XRSTOR instructions for YMM state management. The XSAVE/XRSTOR instructions also provide flexible and efficient interface to manage XMM/MXCSR states and x87 FPU states in conjunction with new processor extended states.

An OS must enable its YMM state management to support AVX and FMA extensions. Otherwise, an attempt to execute an instruction in AVX or FMA extensions (including an enhanced 128-bit SIMD instructions using VEX encoding) will cause a #UD exception.

3.2.1 Detection of YMM State Support

Detection of hardware support for new processor extended state is provided by 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.

3.2.2 Enabling of YMM State

An OS can enable YMM state support with the following steps:

- Verify the processor supports XSAVE/XRSTOR/XSETBV/XGETBV instructions and the XFEATURE_ENABLED_MASK register by checking CPUID.1.ECX.XSAVE[bit 26]=1.
- Verify the processor supports YMM state (i.e. bit 2 of XFEATURE_ENABLED_MASK is valid) by checking CPUID.(EAX=0DH, ECX=0):EAX.YMM[2]. The OS should also verify CPUID.(EAX=0DH, ECX=0):EAX.SSE[bit 1]=1, because the lower 128-bits of an YMM register are aliased to an XMM register.

The OS must determine the buffer size requirement for the XSAVE area that will be used by XSAVE/XRSTOR (see CPUID instruction in Section 2.9).

- Set CR4.OSXSAVE[bit 18]=1 to enable the use of XSETBV/XGETBV instructions to write/read the XFEATURE_ENABLED_MASK 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 x87 FPU, SSE and YMM state management using XSAVE/XRSTOR, the enable mask is EDX=0H, EAX=7H (The individual bits of XFEATURE_ENABLED_MASK is listed in Table 3-1).

To enable YMM state, the OS must use `EDX:EAX[2:1] = 11B` when executing `XSETBV`. An attempt to execute `XSETBV` with `EDX:EAX[2:1] = 10B` causes a `#GP(0)` exception.

Table 3-1. XFEATURE_ENABLED_MASK and Processor State Components

| Bit | Meaning |
|---------|---|
| 0 - x87 | If set, the processor supports x87 FPU state management via <code>XSAVE/XRSTOR</code> . This bit must be 1 if <code>CPUID.01H:ECX.XSAVE[26] = 1</code> . |
| 1 - SSE | If set, the processor supports SSE state (XMM and MXCSR) management via <code>XSAVE/XRSTOR</code> . This bit must be set to '1' to enable AVX. |
| 2 - YMM | If set, the processor supports YMM state (upper 128 bits of YMM registers) management via <code>XSAVE</code> . This bit must be set to '1' to enable AVX and FMA. |

3.2.3 Enabling of SIMD Floating-Exception Support

AVX and FMA instruction 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 and FMA enabling is listed in Table 3-2

Table 3-2. CR4 bits for AVX New Instructions technology support

| Bit | Meaning |
|-------------------------------------|--|
| <code>CR4.OSXSAVE[bit 18]</code> | If set, the OS supports use of <code>XSETBV/XGETBV</code> instruction to access the <code>XFEATURE_ENABLED_MASK</code> register, <code>XSAVE/XRSTOR</code> to manage processor extended state. Must be set to '1' to enable AVX and FMA. |
| <code>CR4.OSXMMEXCPT[bit 10]</code> | Must be set to 1 to enable SIMD floating-point exceptions. This applies to AVX, FMA operating on YMM states, and legacy 128-bit SIMD floating-point instructions operating on XMM states. |
| <code>CR4.OSFXSR[bit 9]</code> | Ignored by AVX and FMA instructions operating on YMM states. Must be set to 1 to enable SIMD instructions operating on XMM state. |

3.2.4 The Layout of XSAVE 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 XFEATURE_ENABLED_MASK, it can allocate the buffer size according to CPUID.(EAX=0DH, ECX=0):ECX.

After the memory buff for XSAVE is allocated, the entire buffer must to cleared to zero prior to use by XSAVE.

For processors that support SSE and YMM states, the XSAVE area layout 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.

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) | CPUID.(EAX=0DH, ECX=2):EBX | CPUID.(EAX=0DH, ECX=2):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] contains 16 of the upper 128-bits of the YMM registers, it is shown in Table 3-5.

Note in general that the layout of the XSAVE/XRSTOR save area is fixed and may contain non-contiguous individual save area (Ext_Save_Area_X). The XSAVE/XRSTOR area is not compacted if some processor extended state features are not saved or are not supported by the processor and/or by system software.

Table 3-5. XSAVE Save Area Layout for YMM State (Ext_Save_Area_2)

| 31 16 | 15 0 | Byte Offset from YMM_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 |

3.2.5 XSAVE/XRSTOR Interaction with YMM State and MXCSR

The processor's action as a result of executing XRSTOR, on the MXCSR, XMM and YMM registers, are listed in Table 3-6 (Both bit 1 and bit 2 of the XFEATURE_ENABLED_MASK register are presumed to be 1). 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. The saving/restoring of MXCSR is bound to both the SSE state and YMM state.

Table 3-6. XRSTOR Action on MXCSR, XMM Registers, YMM Registers

| EDX:EAX | | XSATE_BV | | MXCSR | YMM_H 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 processor supplied init values for each processor state component used by XRSTOR is listed in Table 3-7.

Table 3-7. 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 ¹ | If 64-bit Mode: XMM0-XMM15 ← 0H; Else XMM0-XMM7 ← 0H |
| YMM State ¹ | If 64-bit Mode: YMM0_H-YMM15_H ← 0H; Else YMM0_H-YMM7_H ← 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.

The action of XSAVE is listed in Table 3-8.

Table 3-8. XSAVE Action on MXCSR, XMM, YMM Register

| EDX:EAX | | XFEATURE_ENABL ED_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 |

3.3 RESET BEHAVIOR

At processor reset

- YMM0-16 bits[255:0] are set to zero.

- XFEATURE_ENABLED_MASK[2:1] is set to zero, XFEATURE_ENABLED_MASK[0] is set to 1.
- CR4.OSXSAVE[bit 18] (and its mirror CPUID.1.ECX.OSXSAVE[bit 27]) is set to 0.

3.4 EMULATION

Setting the CR0.EMbit 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 XFEATURE_ENABLED_MASK[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 AVX FLOATING-POINT EXCEPTION HANDLERS

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.OSXMMEXCPT flag (bit 10) must be set.

This page was
intentionally left
blank.

Chapter 4

Instruction Format

AVX and FMA instructions are encoded using a more efficient format than previous instruction extensions in the Intel 64 and IA-32 architecture. The improved encoding format make use a new prefix referred to as “VEX”. The VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the length of the VEX prefix, the instruction encoding format using VEX addresses two important issues: (a) there exists inefficiency in instruction encoding due to SIMD prefixes and some fields of the REX prefix, (b) Both SIMD prefixes and REX prefix increase in instruction byte-length. This chapter describes the instruction encoding format using VEX.

4.1 INSTRUCTION FORMATS

Legacy instruction set extensions in IA-32 architecture employs one or more “single-purpose” byte as an “escape opcode”, or required SIMD prefix (66H, F2H, F3H) to expand the processing capability of the instruction set. Intel 64 architecture uses the REX prefix to expand the encoding of register access in instruction operands. Both SIMD prefixes and REX prefix carry the side effect that they can cause the length of an instruction to increase significantly. Legacy Intel 64 and IA-32 instruction set are limited to supporting instruction syntax of only two operands that can be encoded to access registers (and only one can access a memory address).

Instruction encoding using VEX prefix provides several advantages:

- Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by FMA instructions and operations like VPERMIL2PS is encoded using bits 7:4 of the immediate byte.
- Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers)
- Encoding support for instruction syntax of non-destructive source operands.
- Elimination of escape opcode byte (OFH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

INSTRUCTION FORMAT

Figure 4-1 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, “single-purpose” byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

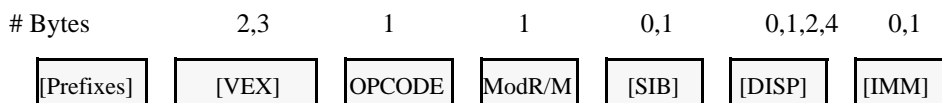


Figure 4-1. Instruction Encoding Format with VEX Prefix

4.1.1 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

4.1.2 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

4.1.3 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix preceding VEX will #UD.

4.1.4 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 4-2.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- Non-destructive source register encoding (applicable to three and four operand syntax): This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1's

complement form (inverted form), i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.

- Vector length encoding: This 1-bit field represented by the notation VEX.L. L=0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.
- REX prefix functionality: Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1's complement form, i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
 - Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1's complement encoding. This is represented as VEX.R.
 - Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1's complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
 - Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.

- Source operand swizzle encoding: Instructions supporting four operands have three source operands and only one of the source operand can be encoded to access a SIMD register or a memory address (via the r/m field of the ModR/M byte). VEX.W can provide source operand swizzling to allow the programmers to use the ModR/M: r/m field to encode either the second or the third source operand.
- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

INSTRUCTION FORMAT

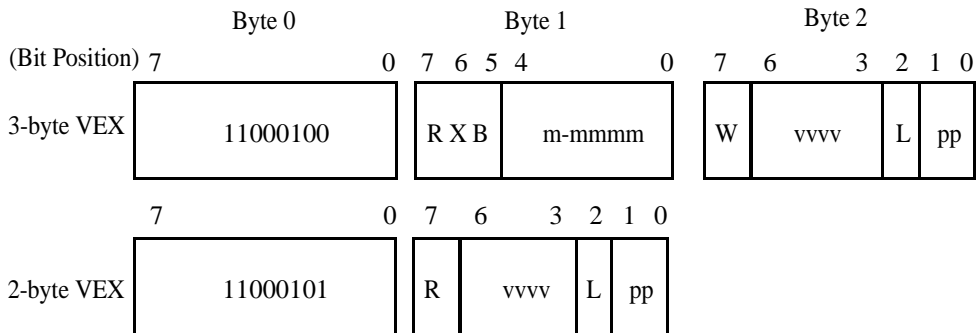
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Some additional support for 128-bit vector integer instructions is provided in Table A-1 of Appendix A. Note, certain new instruction functionality can only be encoded with the VEX prefix (See Appendix A, Table A-2).

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.



R: REX.R in 1's complement (inverted) form

1: Same as REX.R=0 (must be 1 in 32-bit mode)

0: Same as REX.R=1 (64-bit mode only)

X: REX.X in 1's complement (inverted) form

1: Same as REX.X=0 (must be 1 in 32-bit mode)

0: Same as REX.X=1 (64-bit mode only)

B: REX.B in 1's complement (inverted) form

1: Same as REX.B=0 (Ignored in 32-bit mode).

0: Same as REX.B=1 (64-bit mode only)

W: opcode specific (use like REX.W, or used for memory operand select on 4-operand instructions, or ignored, depending on the opcode)

m-mmmm:

00000: Reserved for future use (will #UD)

00001: implied 0F leading opcode byte

00010: implied 0F 38 leading opcode bytes

00011: implied 0F 3A leading opcode bytes

00100-11111: Reserved for future use (will #UD)

vvvv: a register specifier (in 1's complement form) or 1111 if unused.

L: Vector Length

0: scalar or 128-bit vector

1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix

00: None

01: 66

10: F3

11: F2

Figure 4-2. VEX bitfields

The following subsections describe the various fields in two or three-byte VEX prefix:

4.1.4.1 VEX Byte 0, bits[7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

4.1.4.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is present in both 2- and 3-byte VEX prefixes.

The usage of WRXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer's manual, Volume 2A.

This bit is stored in bit inverted format.

4.1.4.3 3-byte VEX byte 1, bit[6] - 'X'

Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

4.1.4.4 3-byte VEX byte 1, bit[5] - 'B'

Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

4.1.4.5 3-byte VEX byte 2, bit[7] - 'W'

Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.

- For AVX instructions that have equivalent legacy SSE instructions, if REX.W has a meaning in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes, VEX.W must be set to "0" otherwise the AVX form will #UD.
- For AVX instructions that have equivalent legacy SSE instructions, if REX.W is

don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.

- For new AVX instructions where VEX.W has no defined function, it is reserved as zero and setting to other than zero will cause instruction to #UD.
- For some new AVX and FMA instructions, VEX.W may be considered as an opcode extension. E.g. 3-source instructions (FMA, VPERMIL2PS, and VPERMIL2PD) use the VEX.W to swap the source operands.

4.1.4.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or dest Register Specifier

In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits.

The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1's complement) form.

VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD.

In 64-bit mode all 4 bits may be used. See Table 4-1 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).

Table 4-1. VEX.vvvv to register name mapping

| VEX.vvvv | Dest Register | Valid in Legacy/Compatibility 32-bit modes? |
|----------|---------------|---|
| 1111B | XMM0/YMM0 | Valid |
| 1110B | XMM1/YMM1 | Valid |
| 1101B | XMM2/YMM2 | Valid |
| 1100B | XMM3/YMM3 | Valid |
| 1011B | XMM4/YMM4 | Valid |
| 1010B | XMM5/YMM5 | Valid |
| 1001B | XMM6/YMM6 | Valid |
| 1000B | XMM7/YMM7 | Valid |
| 0111B | XMM8/YMM8 | Invalid |
| 0110B | XMM9/YMM9 | Invalid |
| 0101B | XMM10/YMM10 | Invalid |
| 0100B | XMM11/YMM11 | Invalid |
| 0011B | XMM12/YMM12 | Invalid |
| 0010B | XMM13/YMM13 | Invalid |
| 0001B | XMM14/YMM14 | Invalid |
| 0000B | XMM15/YMM15 | Invalid |

The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

4.1.5 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g. VEX-encoded pack shift instructions support one source operand and one destination operand).

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax.

The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1's complement) form and is valid for instructions with 2 or more source operands (see Table 4-3).
- VEX.vvvv encodes the destination register operand, specified in 1's complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 4-2. The notation in the "Opcode" column in Table 4-2 is described in detail in section 5.1.1

- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 4-2. Instructions with a VEX.vvvv destination

| Opcode | Instruction mnemonic |
|----------------------------|--------------------------|
| VEX.NDD.128.66.0F 73 /7 ib | VPSLLDQ xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 73 /3 ib | VPSRLDQ xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 71 /2 ib | VPSRLW xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 72 /2 ib | VPSRLD xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 73 /2 ib | VPSRLQ xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 71 /4 ib | VPSRAW xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 72 /4 ib | VPSRAD xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 71 /6 ib | VPSLLW xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 72 /6 ib | VPSLLD xmm1, xmm2, imm8 |
| VEX.NDD.128.66.0F 73 /6 ib | VPSLLQ xmm1, xmm2, imm8 |

The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves two situations:

- Imm8[7:4] encodes the third source register operand when VEX.W = 0.
- For instruction that support operand swizzling, Imm8[7:4] encodes the second source register operand.

Table 4-3 lists each type of instruction syntax and the instruction operand encoding rule for VEX.vvvv, ModR/M.r/m, ModR/M.reg, and Imm8[7:4]. The “Instruction type” column lists the relationship of the destination operand, the number and types of source operands. The encoding of each operand type to VEX.vvvv, ModR/M.r/m, ModR/M.reg, and Imm8[7:4] is shown in the right-hand column.

Table 4-3. Interpreting VEX.vvvv, reg_field, and rm_field.

| Instruction Type | Behavior | How arguments feed the operation |
|--|--|---|
| xmm/ymm := op(reg, reg/mem) | All Fields Used | ModR/M.reg := op (VEX.vvvv, ModR/M.r/m) |
| xmm/ymm := op(reg, reg) | All Fields Used | ModR/M.r/m := op (VEX.vvvv, ModR/M.reg) |
| xmm/ymm := op (reg, reg/mem, reg) ¹ | All Fields Used | ModR/M.reg := op (VEX.vvvv, ModR/M.r/m, imm8[7:4]) (VEX.W=0) |
| xmm/ymm := op (reg, reg, reg/mem) ¹ | All Fields Used | ModR/M.reg := op (VEX.vvvv, imm8[7:4], ModR/M.r/m) (VEX.W=1) |
| xmm/ymm := op(reg/mem) | VEX.vvvv must be 1111b, otherwise instruction will #UD | ModR/M.reg := op (ModR/M.r/m) |
| xmm/ymm := op(xmm/ymm) | reg_field used for opcode extension | VEX.vvvv := op(ModR/M.r/m) |
| r32/r64 := op(reg/mem) ² | VEX.vvvv must be 1111b, otherwise instruction will #UD | ModR/M.reg := op (ModR/M.r/m) |
| implicit(eflags/r32) := op (reg, reg/mem) | VEX.vvvv must be 1111b, otherwise instruction will #UD | implicit(eflags/r32) := op(ModR/M.reg, ModR/M.r/m) |
| xmm/ymm/mem := op(reg) | VEX.vvvv must be 1111b, otherwise instruction will #UD | ModR/M.r/m := op(ModR/M.reg) |
| r32/r64/mem := op(reg) | VEX.vvvv must be 1111b, othwise instruction will #UD | ModR/M.r/m := op (ModR/M.reg) |
| mem := op(reg) | VEX.vvvv must be 1111b, othwise instruction will #UD | ModR/M.r/m := op (ModR/M.reg) |
| mem := op(reg, reg) ³ | All Fields used | ModR/M.r/m := op(VEX.vvvv, ModR/M.reg,) |
| xmm/ymm := op(reg, mem) ⁴ | All Fields Used | ModR/M.reg := op (VEX.vvvv, ModR/M.r/m) |
| xmm/ymm := op (reg, reg/mem, reg) ⁵ | All Fields Used | ModR/M.reg := op (imm8[7:4], ModR/M.r/m, VEX.vvvv) (VEX.W=0) |
| xmm/ymm := op (reg, reg, reg/mem) ⁵ | All Fields Used | ModR/M.reg := op (imm8[7:4], VEX.vvvv, ModR/M.r/m) (VEX.W=1) |

Note 1: Certain instructions (VPERMIL2PS/PD) use the VEX.W bit to swap the rm_field and imm8[7:4] sources.

Note 2: The instruction `VPEXTRW r32, xmm1, imm (VEX.128.66.0F C5 /r ib)` encodes the destination operand in `ModR/M.reg`.

Note 3: `VMASKMOVPS/PD` store form: `VEX.vvvv` holds the mask register, `reg_field` the src register, and `rm_field` the memory operand.

Note 4: `VMASKMOVPS/PD` load form: `VEX.vvvv` holds the mask register, `rm_field` the memory operand, and `reg_field` the destination register.

Note 5: FMA instructions use the `VEX.W` bit to swap `VEX.vvvv` and the `rm_field` sources.

4.1.5.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

Table 4-4. VEX.m-mmmm interpretation

| VEX.m-mmmm | Implied Leading Opcode Bytes |
|--------------|------------------------------|
| 00000B | Reserved |
| 00001B | 0F |
| 00010B | 0F 38 |
| 00011B | 0F 3A |
| 00100-11111B | Reserved |
| (2-byte VEX) | 0F |

`VEX.m-mmmm` is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

4.1.5.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- “L”

The vector length field, `VEX.L`, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If “`VEX.L = 1`”, it indicates 256-bit vector operation. “`VEX.L = 0`” indicates scalar and 128-bit vector operations.

The instruction `VZEROUPPER` is a special case that is encoded with `VEX.L = 0`, although its operation zero’s bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

Table 4-5. VEX.L interpretation

| VEX.L | Vector Length |
|-------|-------------------------------|
| 0 | 128-bit (or 32/64-bit scalar) |
| 1 | 256-bit |

4.1.5.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- “pp”

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.

Table 4-6. VEX.pp interpretation

| pp | Implies this prefix after other prefixes but before VEX |
|-----|---|
| 00B | None |
| 01B | 66 |
| 10B | F3 |
| 11B | F2 |

4.1.6 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

4.1.7 The MODRM, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg_field or rm_field differs (see above).

4.1.8 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support three source operands in a four operand syntax. Generally, the third source operand is encoded in imm8[7:4]. VBLENDVPD, VBLENDVPS, PBLENDVB, VPERMIL2PS and VPERMIL2PD, FMA instructions use imm8[7:4] to encode one of the source registers. When the upper 4 bits of the immediate byte is used for a third source, the definition of the contents of the lower 4 bits is instruction dependent.

A subset of the instruction supporting three source operands may also support swizzled operand encoding using the VEX.W field (see Section 4.1.4.5).

4.1.9 AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper 128 bits of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper 128-bits.

4.1.10 AVX Instruction Length

The AVX and FMA instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

This page was
intentionally left
blank.

CHAPTER 5 INSTRUCTION SET REFERENCE

Instructions that are 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 5.2* 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:

VPERMIL2PS- Permute 2 Source Single-Precision Floating-Point Values (THIS IS AN EXAMPLE)

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.256.66.OF3A.W1 48 /r /is4 VPERMIL2PS ymm1, ymm2, ymm3, ymm4/m256, imz2 | V/V | AVX | Permute single-precision floating-point values in ymm2 and ymm3 using controls from ymm4/mem, the results are stored in ymm1 with selective zero-match controls. |

5.1.2 Opcode Column in the Instruction Summary Table

For notation and conventions applicable to instructions that do not use VEX prefix, 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].[128,256].[66,F2,F3].OF/OF3A/OF38.[W0,W1] 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. Refer to Section 4.1.4 for more detail on the VEX prefix

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

- **NDS, NDD:** 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). If both NDS and NDD absent (i.e. VEX.vvvv does not encode an operand), VEX.vvvv must be 1111b. The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.
- **128,256:** VEX.L field can be 0 (denoted by VEX.128) or 1 (denoted by VEX.256). 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.
- **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 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 consider as an implies 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.
- The presence of W0/W1 in the opcode column applies to two situations: (a) the instruction supports source operand swizzling between two of the three

source operands, (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. Please see Section 4.1.4 on the subfield definitions within VEX.

- **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 providing 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. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 4.1.4.

5.1.3 Instruction Column in the Instruction Summary Table

<additions to the eponymous PRM section>

- **ymm** — a YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode.
- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX and FMA instructions.
- **ymm/m256** - a YMM register or 256-bit memory operand.
- **<YMM0>**: indicates use of the YMM0 register as an implicit argument.
- **SRC1** - Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX prefix and having two or more source operands.
- **SRC2** - Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX prefix and having two or more source operands.
- **SRC3** - Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX prefix and having three source operands.
- **SRC** - The source in a AVX single-source instruction or the source in a Legacy SSE instruction.
- **DST** - the destination in a AVX instruction. In Legacy SSE instructions can be either the destination, first source, or both. 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 (a) 64-bit mode or (b) the Compatibility mode and other IA-32 modes that apply in conjunction with the CPUID feature flag associated specific instruction extensions.

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 bit in CPUID.1.ECX, CPUID.1.EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AVX support) that indicate processor support for the instruction. If the corresponding flag is ‘0’, the instruction will #UD.

5.2 AES TRANSFORMATIONS AND DATA STRUCTURE

5.2.1 Little-Endian Architecture and Big-Endian Specification (FIPS 197)

FIPS 197 document defines the Advanced Encryption Standard (AES) and includes a set of test vectors for testing all of the steps in the algorithm, and can be used for testing and debugging.

The following observation is important for using the AES instructions offered in Intel 64 Architecture: FIPS 197 text convention is to write hex strings with the low-memory byte on the left and the high-memory byte on the right. Intel's convention is the reverse. It is similar to the difference between Big Endian and Little Endian notations.

In other words, a 128 bits vector in the FIPS document, when read from left to right, is encoded as [7:0, 15:8, 23:16, 31:24, ...127:120]. Note that inside the byte, the encoding is [7:0], so the first bit from the left is the most significant bit. In practice, the test vectors are written in hexadecimal notation, where pairs of hexadecimal digits define the different bytes. To translate the FIPS 197 notation to an Intel 64 architecture compatible ("Little Endian") format, each test vector needs to be byte-reflected to [127:120,... 31:24, 23:16, 15:8, 7:0].

Example A:

FIPS Test vector: 0x000102030405060708090a0b0c0d0e0f

Intel AES Hardware: 0x0f0e0d0c0b0a09080706050403020100

It should be pointed out that the only thing at issue is a textual convention, and programmers do not need to perform byte-reversal in their code, when using the AES instructions.

5.2.1.1 AES Data Structure in Intel 64 Architecture

The AES instructions that are defined in this document operate on one or on two 128 bits source operands: State and Round Key. From the architectural point of view, the state is input in an xmm register and the Round key is input either in an xmm register or a 128-bit memory location.

In AES algorithm, the state (128 bits) can be viewed as 4 32-bit doublewords ("Word"s in AES terminology): X3, X2, X1, X0.

The state may also be viewed as a set of 16 bytes. The 16 bytes can also be viewed as a 4x4 matrix of bytes where S(i, j) with i, j = 0, 1, 2, 3 compose the 32-bit "word"s as follows:

X0 = S (3, 0) S (2, 0) S (1, 0) S (0, 0)

X1 = S (3, 1) S (2, 1) S (1, 1) S (0, 1)

$X2 = S(3, 2) S(2, 2) S(1, 2) S(0, 2)$

$X3 = S(3, 3) S(2, 3) S(1, 3) S(0, 3)$

The following tables, Table 5-1 through Table 5-4, illustrate various representations of a 128-bit state.

Table 5-1. Byte and 32-bit Word Representation of a 128-bit State

| | | | | | | | | | | | | | | | | |
|--------------|----------|------|------|-----|---------|----|----|----|---------|----|----|----|--------|----|----|-------|
| Byte # | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Bit Position | 127 | 119 | 111 | 103 | 95 | 87 | 79 | 71 | 63 | 55 | 47 | 39 | 31 | 23 | 15 | 7 - 0 |
| | -120 | -112 | -103 | -96 | | | | | | | | | | | | |
| | 127 - 96 | | | | 95 - 64 | | | | 64 - 32 | | | | 31 - 0 | | | |
| State Word | X3 | | | | X2 | | | | X1 | | | | X0 | | | |
| State Byte | P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A |

Table 5-2. Matrix Representation of a 128-bit State

| | | | | | | | |
|---|---|---|---|---------|---------|---------|---------|
| A | E | I | M | S(0, 0) | S(0, 1) | S(0, 2) | S(0, 3) |
| B | F | J | N | S(1, 0) | S(1, 1) | S(1, 2) | S(1, 3) |
| C | G | K | O | S(2, 0) | S(2, 1) | S(2, 2) | S(2, 3) |
| D | H | L | P | S(3, 0) | S(3, 1) | S(3, 2) | S(3, 3) |

Example:

FIPS vector: d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5

This vector has the “least significant” byte d4 and the significant byte e5 (written in Big Endian format in the FIPS document). When it is translated to IA notations, the encoding is:

Table 5-3. Little Endian Representation of a 128-bit State

| | | | | | | | | | | | | | | | | |
|-------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Byte # | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| State Byte | P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A |
| State Value | e5 | 98 | 27 | 1e | f1 | 11 | 41 | b8 | ae | 52 | b4 | e0 | 30 | 5d | bf | d4 |

Table 5-4. Little Endian Representation of a 4x4 Byte Matrix

| | | | | | | | |
|---|---|---|---|----|----|----|----|
| A | E | I | M | d4 | e0 | b8 | 1e |
| B | F | J | N | bf | b4 | 41 | 27 |
| C | G | K | O | 5d | 52 | 11 | 98 |

Table 5-4. Little Endian Representation of a 4x4 Byte Matrix

| | | | | | | | | |
|---|---|---|---|--|----|----|----|----|
| D | H | L | P | | 30 | ae | f1 | e5 |
|---|---|---|---|--|----|----|----|----|

5.2.2 AES Transformations and Functions

The following functions and transformations are used in the algorithmic descriptions of AES instruction extensions AESDEC, AESDECLAST, AESENC, AESENCLAST, AESIMC, AESKEYGENASSIST.

Note that these transformations are expressed here in a Little Endian format (and not as in the FIPS 197 document).

- **MixColumns():** A byte-oriented 4x4 matrix transformation on the matrix representation of a 128-bit AES state. A FIPS-197 defined 4x4 matrix is multiplied to each 4x1 column vector of the AES state. The columns are considered polynomials with coefficients in the Finite Field that is used in the definition of FIPS 197, the operations (“multiplication” and “addition”) are in that Finite Field, and the polynomials are reduced modulo x^4+1 .

The MixColumns() transformation defines the relationship between each byte of the result state, represented as $S'(i, j)$ of a 4x4 matrix (see Section 5.2.1), as a function of input state bytes, $S(i, j)$, as follows

$$S'(0, j) \leftarrow \text{FF_MUL}(02\text{H}, S(0, j)) \text{ XOR } \text{FF_MUL}(03\text{H}, S(1, j)) \text{ XOR } S(2, j) \text{ XOR } S(3, j)$$

$$S'(1, j) \leftarrow S(0, j) \text{ XOR } \text{FF_MUL}(02\text{H}, S(1, j)) \text{ XOR } \text{FF_MUL}(03\text{H}, S(2, j)) \text{ XOR } S(3, j)$$

$$S'(2, j) \leftarrow S(0, j) \text{ XOR } S(1, j) \text{ XOR } \text{FF_MUL}(02\text{H}, S(2, j)) \text{ XOR } \text{FF_MUL}(03\text{H}, S(3, j))$$

$$S'(3, j) \leftarrow \text{FF_MUL}(03\text{H}, S(0, j)) \text{ XOR } S(1, j) \text{ XOR } S(2, j) \text{ XOR } \text{FF_MUL}(02\text{H}, S(3, j))$$

where $j = 0, 1, 2, 3$. $\text{FF_MUL}(\text{Byte1}, \text{Byte2})$ denotes the result of multiplying two elements (represented by Byte1 and Byte2) in the Finite Field representation that defines AES. The result of produced by $\text{FF_MUL}(\text{Byte1}, \text{Byte2})$ is an element in the Finite Field (represented as a byte). A Finite Field is a field with a finite number of elements, and when this number can be represented as a power of 2 (2^n), its elements can be represented as the set of 2^n binary strings of length n . AES uses a finite field with $n=8$ (having 256 elements). With this representation, “addition” of two elements in that field is a bit-wise XOR of their binary-string representation, producing another element in the field. Multiplication of two elements in that field is defined using an irreducible polynomial (for AES, this polynomial is $m(x) = x^8 + x^4 + x^3 + x + 1$). In this Finite Field representation, the bit value of bit position k of a byte represents the coefficient of a polynomial of order k , e.g., 1010_1101B (ADH) is represented by the polynomial $(x^7 + x^5 + x^3 + x^2 + 1)$. The byte value result of multiplication of two elements is obtained by a carry-less multiplication of the two corresponding polynomials, followed by reduction modulo the polynomial, where the remainder

is calculated using operations defined in the field. For example, $FF_MUL(57H, 83H) = C1H$, because the carry-less polynomial multiplication of the polynomials represented by 57H and 83H produces $(x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1)$, and the remainder modulo $m(x)$ is $(x^7 + x^6 + 1)$.

- **RotWord()**: performs a byte-wise cyclic permutation (rotate right in little-endian byte order) on a 32-bit AES word.

The output word $X'[j]$ of $RotWord(X[j])$ where $X[j]$ represent the four bytes of column j , $S(i, j)$, in descending order $X[j] = (S(3, j), S(2, j), S(1, j), S(0, j))$; $X'[j] = (S'(3, j), S'(2, j), S'(1, j), S'(0, j)) \leftarrow (S(0, j), S(3, j), S(2, j), S(1, j))$

- **ShiftRows()**: A byte-oriented matrix transformation that processes the matrix representation of a 16-byte AES state by cyclically shifting the last three rows of the state by different offset to the left, see Figure 5-5.

Table 5-5. The ShiftRows Transformation

| Matrix Representation of Input State | | | | Output of ShiftRows | | | |
|--------------------------------------|---|---|---|---------------------|---|---|---|
| A | E | I | M | A | E | I | M |
| B | F | J | N | F | J | N | B |
| C | G | K | O | K | O | C | G |
| D | H | L | P | P | D | H | L |

- **SubBytes()**: A byte-oriented transformation that processes the 128-bit AES state by applying a non-linear substitution table (S-BOX) on each byte of the state.

The **SubBytes()** function defines the relationship between each byte of the result state $S'(i, j)$ as a function of input state byte $S(i, j)$, by

$$S'(i, j) \leftarrow S\text{-Box}(S(i, j)[7:4], S(i, j)[3:0])$$

where $S\text{-BOX}(S[7:4], S[3:0])$ represents a look-up operation on a 16x16 table to return a byte value, see Table 5-6.

Table 5-6. Look-up Table Associated with S-Box Transformation

| | | S[3:0] | | | | | | | | | | | | | | | |
|--------|---|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
| S[7:4] | 0 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| | 1 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72 | c0 |
| | 2 | b7 | fd | 93 | 26 | 36 | 3f | f7 | cc | 34 | a5 | e5 | f1 | 71 | d8 | 31 | 15 |
| | 3 | 04 | c7 | 23 | c3 | 18 | 96 | 05 | 9a | 07 | 12 | 80 | e2 | eb | 27 | b2 | 75 |
| | 4 | 09 | 83 | 2c | 1a | 1b | 6e | 5a | a0 | 52 | 3b | d6 | b3 | 29 | e3 | 2f | 84 |
| | 5 | 53 | d1 | 00 | ed | 20 | fc | b1 | 5b | 6a | cb | be | 39 | 4a | 4c | 58 | cf |
| | 6 | d0 | ef | aa | fb | 43 | 4d | 33 | 85 | 45 | f9 | 02 | 7f | 50 | 3c | 9f | a8 |
| | 7 | 51 | a3 | 40 | 8f | 92 | 9d | 38 | f5 | bc | b6 | da | 21 | 10 | ff | f3 | d2 |
| | 8 | cd | 0c | 13 | ec | 5f | 97 | 44 | 17 | c4 | a7 | 7e | 3d | 64 | 5d | 19 | 73 |
| | 9 | 60 | 81 | 4f | dc | 22 | 2a | 90 | 88 | 46 | ee | b8 | 14 | de | 5e | 0b | db |
| | a | e0 | 32 | 3a | 0a | 49 | 06 | 24 | 5c | c2 | d3 | ac | 62 | 91 | 95 | e4 | 79 |
| | b | e7 | c8 | 37 | 6d | 8d | d5 | 4e | a9 | 6c | 56 | f4 | ea | 65 | 7a | ae | 08 |
| | c | ba | 78 | 25 | 2e | 1c | a6 | b4 | c6 | e8 | dd | 74 | 1f | 4b | bd | 8b | 8a |
| | d | 70 | 3e | b5 | 66 | 48 | 03 | f6 | 0e | 61 | 35 | 57 | b9 | 86 | c1 | 1d | 9e |
| | e | e1 | f8 | 98 | 11 | 69 | d9 | 8e | 94 | 9b | 1e | 87 | e9 | ce | 55 | 28 | df |
| | f | 8c | a1 | 89 | 0d | bf | e6 | 42 | 68 | 41 | 99 | 2d | 0f | b0 | 54 | bb | 16 |

- SubWord(): produces an output AES word (four bytes) from the four bytes of an input word using a non-linear substitution table (S-BOX).

$$X'[j] = (S'(3, j), S'(2, j), S'(1, j), S'(0, j)) \leftarrow (S\text{-Box}(S(3, j)), S\text{-Box}(S(2, j)), S\text{-Box}(S(1, j)), S\text{-Box}(S(0, j)))$$

- InvMixColumns(): The inverse transformation of MixColumns().

The InvMixColumns() transformation defines the relationship between each byte of the result state $S'(i, j)$ as a function of input state bytes, $S(i, j)$, by

$$S'(0, j) \leftarrow FF_MUL(0eH, S(0, j)) \text{ XOR } FF_MUL(0bH, S(1, j)) \text{ XOR } FF_MUL(0dH, S(2, j)) \text{ XOR } FF_MUL(09H, S(3, j))$$

$$S'(1, j) \leftarrow FF_MUL(09H, S(0, j)) \text{ XOR } FF_MUL(0eH, S(1, j)) \text{ XOR } FF_MUL(0bH, S(2, j)) \text{ XOR } FF_MUL(0dH, S(3, j))$$

$$S'(2, j) \leftarrow FF_MUL(0dH, S(0, j)) \text{ XOR } FF_MUL(09H, S(1, j)) \text{ XOR } FF_MUL(0eH, S(2, j)) \text{ XOR } FF_MUL(0bH, S(3, j))$$

$S'(3, j) \leftarrow \text{FF_MUL}(0bH, S(0, j)) \text{ XOR } \text{FF_MUL}(0dH, S(1, j)) \text{ XOR } \text{FF_MUL}(09H, S(2, j)) \text{ XOR } \text{FF_MUL}(0eH, S(3, j))$, where $j = 0, 1, 2, 3$.

- `InvShiftRows()`: The inverse transformation of `InvShiftRows()`. The `InvShiftRows()` transforms the matrix representation of a 16-byte AES state by cyclically shifting the last three rows of the state by different offset to the right, see Table 5-7.

Table 5-7. The `InvShiftRows` Transformation

| Matrix Representation of Input State | | | | Output of ShiftRows | | | |
|--------------------------------------|---|---|---|---------------------|---|---|---|
| A | E | I | M | A | E | I | M |
| B | F | J | N | N | B | F | J |
| C | G | K | O | K | O | C | G |
| D | H | L | P | H | L | P | D |

- `InvSubBytes()`: The inverse transformation of `SubBytes()`.
The `InvSubBytes()` transformation defines the relationship between each byte of the result state $S'(i, j)$ as a function of input state byte $S(i, j)$, by
 $S'(i, j) \leftarrow \text{InvS-Box}(S(i, j)[7:4], S(i, j)[3:0])$
 where `InvS-BOX(S[7:4], S[3:0])` represents a look-up operation on a 16x16 table to return a byte value, see Table 5-8.

Table 5-8. Look-up Table Associated with InvS-Box Transformation

| | | S[3:0] | | | | | | | | | | | | | | | |
|--------|---|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
| S[7:4] | 0 | 52 | 09 | 6a | d5 | 30 | 36 | a5 | 38 | bf | 40 | a3 | 9e | 81 | f3 | d7 | fb |
| | 1 | 7c | e3 | 39 | 82 | 9b | 2f | ff | 87 | 34 | 8e | 43 | 44 | c4 | de | e9 | cb |
| | 2 | 54 | 7b | 94 | 32 | a6 | c2 | 23 | 3d | ee | 4c | 95 | 0b | 42 | fa | c3 | 4e |
| | 3 | 08 | 2e | a1 | 66 | 28 | d9 | 24 | b2 | 76 | 5b | a2 | 49 | 6d | 8b | d1 | 25 |
| | 4 | 72 | f8 | f6 | 64 | 86 | 68 | 98 | 16 | d4 | a4 | 5c | cc | 5d | 65 | b6 | 92 |
| | 5 | 6c | 70 | 48 | 50 | fd | ed | b9 | da | 5e | 15 | 46 | 57 | a7 | 8d | 9d | 84 |
| | 6 | 90 | d8 | ab | 00 | 8c | bc | d3 | 0a | f7 | e4 | 58 | 05 | b8 | b3 | 45 | 06 |
| | 7 | d0 | 2c | 1e | 8f | ca | 3f | 0f | 02 | c1 | af | bd | 03 | 01 | 13 | 8a | 6b |
| | 8 | 3a | 91 | 11 | 41 | 4f | 67 | dc | ea | 97 | f2 | cf | ce | f0 | b4 | e6 | 73 |
| | 9 | 96 | ac | 74 | 22 | e7 | ad | 35 | 85 | e2 | f9 | 37 | e8 | 1c | 75 | df | 6e |
| | a | 47 | f1 | 1a | 71 | 1d | 29 | c5 | 89 | 6f | b7 | 62 | 0e | aa | 18 | be | 1b |
| | b | fc | 56 | 3e | 4b | c6 | d2 | 79 | 20 | 9a | db | c0 | fe | 78 | cd | 5a | f4 |
| | c | 1f | dd | a8 | 33 | 88 | 07 | c7 | 31 | b1 | 12 | 10 | 59 | 27 | 80 | ec | 5f |
| | d | 60 | 51 | 7f | a9 | 19 | b5 | 4a | 0d | 2d | e5 | 7a | 9f | 93 | c9 | 9c | ef |
| | e | a0 | e0 | 3b | 4d | ae | 2a | f5 | b0 | c8 | eb | bb | 3c | 83 | 53 | 99 | 61 |
| | f | 17 | 2b | 04 | 7e | ba | 77 | d6 | 26 | e1 | 69 | 14 | 63 | 55 | 21 | 0c | 7d |

5.3 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 prefix.
- **XGETBV, XSETBV, XSAVE, XRSTOR** are defined in *IA-32 Intel Architecture 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.
- **VEX.vvvv**. The VEX bitfield 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

5.4 INSTRUCTION SET REFERENCE

<only instructions modified by AVX are included>

ADDPD - Add Packed Double Precision Floating-Point Values

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

Description

Performs an SIMD add of the two or four 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.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDPD (VEX.256 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$$

$$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] + \text{SRC2}[191:128]$$

$$\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] + \text{SRC2}[255:192]$$

VADDPD (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$
ADDPD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] + \text{SRC}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]$$

$$\text{DEST}[255:128] \text{ (Unmodified)}$$
Intel C/C++ Compiler Intrinsic Equivalent

VADDPD `__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

See Exceptions Type 2

ADDPS- Add Packed Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 58 /r ADDPS xmm1, xmm2/m128 | V/V | SSE | Add packed single-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1 |
| VEX.NDS.128.0F 58 /r VADDPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Add packed single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1 |
| VEX.NDS.256.0F 58 /r VADDPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Add packed single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1 |

Description

Performs an SIMD add of the four or eight packed single-precision floating-point values from the first Source operand to the Second Source operand, and stores the packed single-precision floating-point results in the destination operand.

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

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDPS (VEX.256 encoded version)

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$$

$$\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$$

$$\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$$

$$\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$$

$\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] + \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] + \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224]$.

VADDPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

ADDPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPS __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

See Exceptions Type 2

ADDSD- Add Scalar Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F2 0F 58 /r ADDSD xmm1, xmm2/m64 | 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 58 /r VADDSD xmm1,xmm2, xmm3/m64 | V/V | AVX | Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1 |

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: Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$

ADDSD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] + \text{SRC}[63:0]$$

$$\text{DEST}[255:64] \text{ (Unmodified)}$$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

ADDSS- Add Scalar Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F3 0F 58 /r ADDSS xmm1, xmm2/m32 | 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 58 /r VADDSS xmm1,xmm2, xmm3/m32 | V/V | AVX | Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1 |

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: Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$$

$$\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$$

$$\text{DEST}[255:128] \leftarrow 0$$

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

$$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] + \text{SRC}[31:0]$$

$$\text{DEST}[255:32] \text{ (Unmodified)}$$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

ADDSUBPD- Packed Double FP Add/Subtract

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

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDSUBPD (VEX.256 encoded version)

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

VADDSUBPD (VEX.128 encoded version)

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

ADDSUBPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDSUBPD `__m256d _mm256_addsub_pd (__m256d a, __m256d b);`

ADDSUBPD `__m128d _mm_addsub_pd (__m128d a, __m128d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

ADDSUBPS- Packed Single FP Add/Subtract

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

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VADDSUBPS (VEX.256 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] - SRC2[31:0]$

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

VADDSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

ADDSUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] - \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{DEST}[63:32] + \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{DEST}[95:64] - \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{DEST}[127:96] + \text{SRC}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDSUBPS `__m256 _mm256_addsub_ps (__m256 a, __m256 b);`

ADDSUBPS `__m128 _mm_addsub_ps (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

AESENC/AESENCLAST- Perform One Round of an AES Encryption Flow

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

Description

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

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

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an

XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESENC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESENC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)
```

VAESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC `__m128i _mm_aesenc (__m128i, __m128i)`

(V)AESENCLAST `__m128i _mm_aesenclast (__m128i, __m128i)`

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4

AESDEC/AESDECLAST- Perform One Round of an AES Decryption Flow

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

Description

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

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

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESDEC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESDEC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)
```

VAESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] ← 0
```

AESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)

(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4

AESIMC- Perform the AES InvMixColumn Transformation

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

Description

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

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

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

Operation

VAESIMC

DEST[127:0] ← InvMixColumns(SRC);

DEST[255:128] ← 0;

AESIMC

DEST[127:0] ← InvMixColumns(SRC);

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC __m128i _mm_aesimc (__m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

AESKEYGENASSIST - AES Round Key Generation Assist

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

Description

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

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

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. [Operation](#)

VAESKEYGENASSIST

$X3[31:0] \leftarrow \text{SRC}[127:96];$

$X2[31:0] \leftarrow \text{SRC}[95:64];$

$X1[31:0] \leftarrow \text{SRC}[63:32];$

$X0[31:0] \leftarrow \text{SRC}[31:0];$

$\text{RCON}[31:0] \leftarrow \text{ZeroExtend}(\text{Imm8}[7:0]);$

$\text{DEST}[31:0] \leftarrow \text{SubWord}(X1);$

$\text{DEST}[63:32] \leftarrow \text{RotWord}(\text{SubWord}(X1)) \text{ XOR } \text{RCON};$

$\text{DEST}[95:64] \leftarrow \text{SubWord}(X3);$

$\text{DEST}[127:96] \leftarrow \text{RotWord}(\text{SubWord}(X3)) \text{ XOR } \text{RCON};$

DEST[255:128] \leftarrow 0;

AESKEYGENASSIST

X3[31:0] \leftarrow SRC [127: 96];

X2[31:0] \leftarrow SRC [95: 64];

X1[31:0] \leftarrow SRC [63: 32];

X0[31:0] \leftarrow SRC [31: 0];

RCON[31:0] \leftarrow ZeroExtend(Imm8[7:0]);

DEST[31:0] \leftarrow SubWord(X1);

DEST[63:32] \leftarrow RotWord(SubWord(X1)) XOR RCON;

DEST[95:64] \leftarrow SubWord(X3);

DEST[127:96] \leftarrow RotWord(SubWord(X3)) XOR RCON;

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 54 /r ANDPD xmm1, xmm2/m128 | 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 VANDPD xmm1,xmm2, xmm3/m128 | 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 VANDPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 54 /r ANDPS xmm1, xmm2/m128 | V/V | SSE | Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem |
| VEX.NDS.128.0F 54 /r VANDPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem |
| VEX.NDS.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

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

VANDPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE AND } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE AND } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE AND } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE AND } \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

ANDPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] \text{ BITWISE AND } \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{DEST}[63:32] \text{ BITWISE AND } \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{DEST}[95:64] \text{ BITWISE AND } \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{DEST}[127:96] \text{ BITWISE AND } \text{SRC}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 55 /r ANDNPD xmm1, xmm2/m128 | 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 VANDNPD xmm1,xmm2, xmm3/m128 | 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 VANDNPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

$\text{DEST}[255:192] \leftarrow (\text{NOT}(\text{SRC1}[255:192])) \text{ BITWISE AND } \text{SRC2}[255:192]$

VANDNPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow (\text{NOT}(\text{SRC1}[63:0])) \text{ BITWISE AND } \text{SRC2}[63:0]$

$\text{DEST}[127:64] \leftarrow (\text{NOT}(\text{SRC1}[127:64])) \text{ BITWISE AND } \text{SRC2}[127:64]$

$\text{DEST}[255:128] \leftarrow 0$

ANDNPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow (\text{NOT}(\text{DEST}[63:0])) \text{ BITWISE AND } \text{SRC}[63:0]$

$\text{DEST}[127:64] \leftarrow (\text{NOT}(\text{DEST}[127:64])) \text{ BITWISE AND } \text{SRC}[127:64]$

$\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 0F 55 /r ANDNPS xmm1, xmm2/m128 | V/V | SSE | Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem |
| VEX.NDS.128.0F 55 /r VANDNPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem |
| VEX.NDS.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

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

VANDNPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow (\text{NOT}(\text{SRC1}[31:0])) \text{ BITWISE AND } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow (\text{NOT}(\text{SRC1}[63:32])) \text{ BITWISE AND } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow (\text{NOT}(\text{SRC1}[95:64])) \text{ BITWISE AND } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow (\text{NOT}(\text{SRC1}[127:96])) \text{ BITWISE AND } \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

ANDNPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow (\text{NOT}(\text{DEST}[31:0])) \text{ BITWISE AND } \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow (\text{NOT}(\text{DEST}[63:32])) \text{ BITWISE AND } \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow (\text{NOT}(\text{DEST}[95:64])) \text{ BITWISE AND } \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow (\text{NOT}(\text{DEST}[127:96])) \text{ BITWISE AND } \text{SRC}[127:96]$
 $\text{DEST}[255:128] \text{ (Unmodified)}$

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4

BLENDPD- Blend Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 3A 0D /r ib BLENDPD xmm1, xmm2/m128, imm8 | V/V | SSE4_1 | Select packed double-precision floating-point Values from xmm1 and xmm2/m128 from mask in imm8 |
| VEX.NDS.128.66.0F3A 0D /r ib VBLENDPD xmm1, xmm2, xmm3/m128, imm8 | V/V | AVX | Select packed double-precision floating-point Values from xmm2 and xmm3/m128 from mask in imm8 |
| VEX.NDS.256.66.0F3A 0D /r ib VBLENDPD ymm1, ymm2, ymm3/m256, imm8 | V/V | AVX | Select packed double-precision floating-point Values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1 |

Description

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

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 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 (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VBLENDPD (VEX.256 encoded version)

```

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

```

VBLENDPD (VEX.128 encoded version)

```

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

```

BLENDPD (128-bit Legacy SSE version)

```

IF (IMM8[0] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDPD `__m256d _mm256_blend_pd (__m256d a, __m256d b, const int mask);`

BLENDPD `__m128d _mm_blend_pd (__m128d a, __m128d b, const int mask);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

BLENDPS- Blend Packed Single Precision Floating-Point Values

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

Description

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

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 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 (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VBLENDPS (VEX.256 encoded version)

IF (IMM8[0] = 0) THEN DEST[31:0] \leftarrow SRC1[31:0]
 ELSE DEST [31:0] \leftarrow SRC2[31:0] FI

IF (IMM8[1] = 0) THEN DEST[63:32] \leftarrow SRC1[63:32]
 ELSE DEST [63:32] \leftarrow SRC2[63:32] FI

IF (IMM8[2] = 0) THEN DEST[95:64] \leftarrow SRC1[95:64]
 ELSE DEST [95:64] \leftarrow SRC2[95:64] FI

IF (IMM8[3] = 0) THEN DEST[127:96] \leftarrow SRC1[127:96]
 ELSE DEST [127:96] \leftarrow SRC2[127:96] FI

IF (IMM8[4] = 0) THEN DEST[159:128] \leftarrow SRC1[159:128]
 ELSE DEST [159:128] \leftarrow SRC2[159:128] FI

IF (IMM8[5] = 0) THEN DEST[191:160] \leftarrow SRC1[191:160]
 ELSE DEST [191:160] \leftarrow SRC2[191:160] FI

IF (IMM8[6] = 0) THEN DEST[223:192] \leftarrow SRC1[223:192]
 ELSE DEST [223:192] \leftarrow SRC2[223:192] FI

IF (IMM8[7] = 0) THEN DEST[255:224] \leftarrow SRC1[255:224]
 ELSE DEST [255:224] \leftarrow SRC2[255:224] FI.

VBLENDPS (VEX.128 encoded version)

IF (IMM8[0] = 0) THEN DEST[31:0] \leftarrow SRC1[31:0]
 ELSE DEST [31:0] \leftarrow SRC2[31:0] FI

IF (IMM8[1] = 0) THEN DEST[63:32] \leftarrow SRC1[63:32]
 ELSE DEST [63:32] \leftarrow SRC2[63:32] FI

IF (IMM8[2] = 0) THEN DEST[95:64] \leftarrow SRC1[95:64]
 ELSE DEST [95:64] \leftarrow SRC2[95:64] FI

IF (IMM8[3] = 0) THEN DEST[127:96] \leftarrow SRC1[127:96]
 ELSE DEST [127:96] \leftarrow SRC2[127:96] FI

DEST[255:128] \leftarrow 0

BLENDPS (128-bit Legacy SSE version)

IF (IMM8[0] = 0) THEN DEST[31:0] \leftarrow DEST[31:0]
 ELSE DEST [31:0] \leftarrow SRC[31:0] FI

IF (IMM8[1] = 0) THEN DEST[63:32] \leftarrow DEST[63:32]
 ELSE DEST [63:32] \leftarrow SRC[63:32] FI

IF (IMM8[2] = 0) THEN DEST[95:64] \leftarrow DEST[95:64]
 ELSE DEST [95:64] \leftarrow SRC[95:64] FI

IF (IMM8[3] = 0) THEN DEST[127:96] \leftarrow DEST[127:96]
 ELSE DEST [127:96] \leftarrow SRC[127:96] FI

DEST[255:128] (Unmodified)

INSTRUCTION SET REFERENCE

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDPS `__m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);`

BLENDPS `__m128 _mm_blend_ps (__m128 a, __m128 b, const int mask);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

BLENDVPD- Blend Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 38 15 /r BLENDVPD xmm1, xmm2/m128, <XMM0> | V/V | SSE4_1 | Conditionally copy double-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the implicit mask operand, XMM0. |
| VEX.NDS.128.66.0F3A 4B /r /is4 VBLENDVPD xmm1, xmm2, xmm3/m128, xmm4 | V/V | AVX | Conditionally copy double-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4 |
| VEX.NDS.256.66.0F3A 4B /r /is4 VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4 | V/V | AVX | Conditionally copy double-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4 |

Description

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

Each quadword element of the destination operand is copied from:

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

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

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (255:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory

location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (255:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

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

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

Operation

VBLENDVPD (VEX.256 encoded version)

MASK \leftarrow SRC3

IF (MASK[63] = 0) THEN DEST[63:0] \leftarrow SRC1[63:0]

ELSE DEST [63:0] \leftarrow SRC2[63:0] FI

IF (MASK[127] = 0) THEN DEST[127:64] \leftarrow SRC1[127:64]

ELSE DEST [127:64] \leftarrow SRC2[127:64] FI

IF (MASK[191] = 0) THEN DEST[191:128] \leftarrow SRC1[191:128]

ELSE DEST [191:128] \leftarrow SRC2[191:128] FI

IF (MASK[255] = 0) THEN DEST[255:192] \leftarrow SRC1[255:192]

ELSE DEST [255:192] \leftarrow SRC2[255:192] FI

VBLENDVPD (VEX.128 encoded version)

MASK \leftarrow SRC3

IF (MASK[63] = 0) THEN DEST[63:0] \leftarrow SRC1[63:0]

ELSE DEST [63:0] \leftarrow SRC2[63:0] FI

IF (MASK[127] = 0) THEN DEST[127:64] \leftarrow SRC1[127:64]

ELSE DEST [127:64] \leftarrow SRC2[127:64] FI

DEST[255:128] \leftarrow 0

BLENDVPD (128-bit Legacy SSE version)

MASK \leftarrow XMM0

IF (MASK[63] = 0) THEN DEST[63:0] \leftarrow DEST[63:0]

ELSE DEST [63:0] \leftarrow SRC[63:0] FI

IF (MASK[127] = 0) THEN DEST[127:64] \leftarrow DEST[127:64]

ELSE DEST [127:64] \leftarrow SRC[127:64] FI

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDVPD __m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);

VBLENDVPD __m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

BLENDVPS- Blend Packed Single Precision Floating-Point Values

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

Description

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

Each quadword element of the destination operand is copied from:

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

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

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (255:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of

the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (255:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

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

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

Operation

VBLENDVPS (VEX.256 encoded version)

MASK \leftarrow SRC3

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

VBLENDVPS (VEX.128 encoded version)

MASK \leftarrow SRC3

```
IF (MASK[31] = 0) THEN DEST[31:0]  $\leftarrow$  SRC1[31:0]
    ELSE DEST [31:0]  $\leftarrow$  SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32]  $\leftarrow$  SRC1[63:32]
    ELSE DEST [63:32]  $\leftarrow$  SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64]  $\leftarrow$  SRC1[95:64]
    ELSE DEST [95:64]  $\leftarrow$  SRC2[95:64] FI
```

INSTRUCTION SET REFERENCE

```
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[255:128] ← 0
```

BLENDVPS (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDVPS __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);

VBLENDVPS __m128 _mm_blendv_ps (__m128 a, __m128 b, __m128 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

VBROADCAST- Load with Broadcast

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| VEX.128.66.0F38 18 /r VBROADCASTSS xmm1, m32 | V/V | AVX | Broadcast single-precision floating-point element in mem to four locations in xmm1 |
| VEX.256.66.0F38 18 /r VBROADCASTSS ymm1, m32 | V/V | AVX | Broadcast single-precision floating-point element in mem to eight locations in ymm1 |
| VEX.256.66.0F38 19 /r VBROADCASTSD ymm1, m64 | V/V | AVX | Broadcast double-precision floating-point element in mem to four locations in ymm1 |
| VEX.256.66.0F38 1A /r VBROADCASTF128 ymm1, m128 | V/V | AVX | Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1 |

Description

Load floating point values from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD.

VBROADCASTSD and VBROADCASTF128 are only supported as 256-bit wide versions. VBROADCASTSS is supported in both 128-bit and 256-bit wide versions.

Note: In VEX-encoded versions, VEX.vvvv is 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.

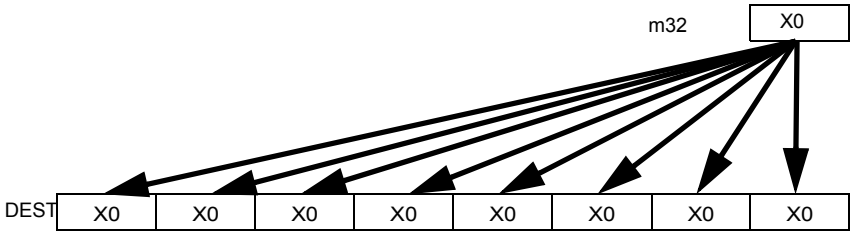


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

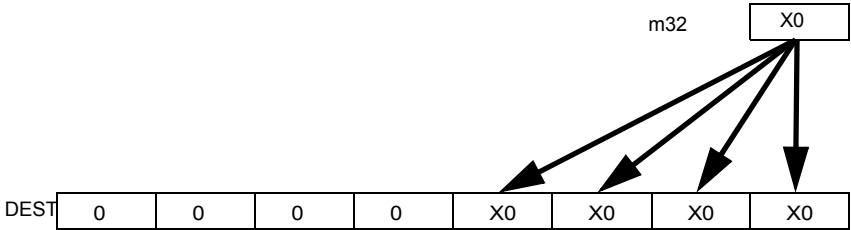


Figure 5-2. VBROADCASTSS Operation (128-bit version)

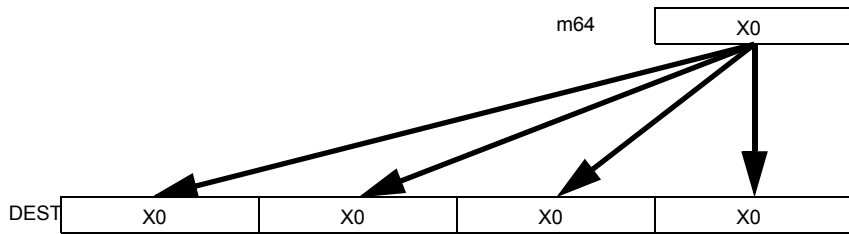


Figure 5-3. VBROADCASTSD Operation

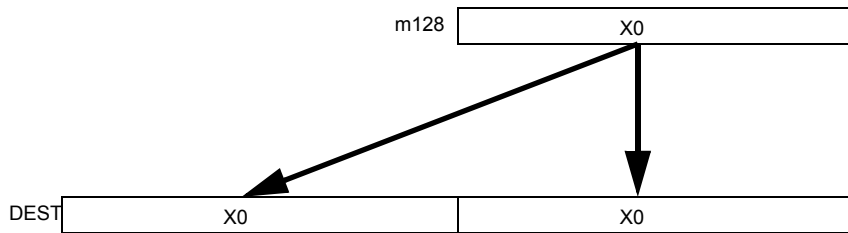


Figure 5-4. VBROADCASTF128 Operation

Operation

VBROADCASTSS (128 bit version)

```

temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[255:128] ← 0

```


INSTRUCTION SET REFERENCE

VBROADCASTSS (VEX.256 encoded version)

temp \leftarrow SRC[31:0]
DEST[31:0] \leftarrow temp
DEST[63:32] \leftarrow temp
DEST[95:64] \leftarrow temp
DEST[127:96] \leftarrow temp
DEST[159:128] \leftarrow temp
DEST[191:160] \leftarrow temp
DEST[223:192] \leftarrow temp
DEST[255:224] \leftarrow temp

VBROADCASTSD (VEX.256 encoded version)

temp \leftarrow SRC[63:0]
DEST[63:0] \leftarrow temp
DEST[127:64] \leftarrow temp
DEST[191:128] \leftarrow temp
DEST[255:192] \leftarrow temp

VBROADCASTF128

temp \leftarrow SRC[127:0]
DEST[127:0] \leftarrow temp
DEST[255:128] \leftarrow temp

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTSS __m128_mm_broadcast_ss(float *a);
VBROADCASTSS __m256_mm256_broadcast_ss(float *a);
VBROADCASTSD __m256d_mm256_broadcast_sd(double *a);
VBROADCASTF128 __m256_mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d_mm256_broadcast_pd(__m128d * a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6, additionally

#UD If VEX.L = 0 for VBROADCASTSD.
 If VEX.L = 0 for VBROADCASTF128

CMPPD- Compare Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8 | 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 C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8 | 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 C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8 | 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 |

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

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 (255:128) of the corresponding YMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand.

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 (255:128) of the destination YMM 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 prefix, bits 4:0 define the type of comparison to be performed (see Table 5-9). 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 5-9). Bits 3 through 7 of the immediate are reserved.

Table 5-9. Comparison Predicate for CMPPD and CMPPS Instructions

| Predicate | imm8 Value | Description | Result: A Is 1st Operand, B Is 2nd Operand | | | | Signals #IA on QNAN |
|-----------------|------------|--|--|-------|-------|------------------------|---------------------|
| | | | A > B | A < B | A = B | Unordered ¹ | |
| EQ_OQ (EQ) | 0H | Equal (ordered, non-signaling) | False | False | True | False | No |
| LT_OS (LT) | 1H | Less-than (ordered, signaling) | False | True | False | False | Yes |
| LE_OS (LE) | 2H | Less-than-or-equal (ordered, signaling) | False | True | True | False | Yes |
| UNORD_Q (UNORD) | 3H | Unordered (non-signaling) | False | False | False | True | No |
| NEQ_UQ (NEQ) | 4H | Not-equal (unordered, non-signaling) | True | True | False | True | No |
| NLT_US (NLT) | 5H | Not-less-than (unordered, signaling) | True | False | True | True | Yes |
| NLE_US (NLE) | 6H | Not-less-than-or-equal (unordered, signaling) | True | False | False | True | Yes |
| ORD_Q (ORD) | 7H | Ordered (non-signaling) | True | True | True | False | No |
| EQ_UQ | 8H | Equal (unordered, non-signaling) | False | False | True | True | No |
| NGE_US (NGE) | 9H | Not-greater-than-or-equal (unordered, signaling) | False | True | False | True | Yes |

Table 5-9. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

| Predicate | imm8 Value | Description | Result: A Is 1st Operand, B Is 2nd Operand | | | | Signals #IA on QNAN |
|------------------|------------|--|--|-------|-------|------------------------|---------------------|
| | | | A > B | A < B | A = B | Unordered ¹ | |
| NGT_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 5-9. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

| Predicate | imm8 Value | Description | Result: A Is 1st Operand, B Is 2nd Operand | | | | Signals #IA on QNaN |
|-----------|------------|---|--|-------|-------|------------------------|---------------------|
| | | | A > B | A < B | A = B | Unordered ¹ | |
| NGE_UQ | 19H | Not-greater-than-or-equal (unordered, nonsignaling) | 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, non-signaling) | True | False | True | False | No |
| GT_OQ | 1EH | Greater-than (ordered, non-signaling) | 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 5-10. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-10. 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> |
| CMPLDPD <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 5-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 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.

Table 5-11. Pseudo-Op and VCMPPD Implementation

| Pseudo-Op | CMPPD Implementation |
|-------------------------------------|-----------------------------------|
| 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> |
| VCMPLDPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 2</i> |
| VCMPUNORDPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 3</i> |
| VCMPNEQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 4</i> |
| VCMPNLTPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 5</i> |
| VCMPNLEPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 6</i> |
| VCMPODPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 7</i> |

Table 5-11. Pseudo-Op and VCMPPD Implementation

| Pseudo-Op | CMPD Implementation |
|--------------------------------------|-------------------------------------|
| VCMPEQ_UQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 8</i> |
| VCMPNGEPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 9</i> |
| VCMPNGTPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 0AH</i> |
| VCMPFALSEPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 0BH</i> |
| VCMPNEQ_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 0CH</i> |
| VCMPGEPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 0DH</i> |
| VCMPGTPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 0EH</i> |
| VCMPTTRUEPD <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> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 12H</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 13H</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 14H</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 15H</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 16H</i> |
| VCMPLT_OQPD <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> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 19H</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 1AH</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 1BH</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 1CH</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 1DH</i> |
| VCMPLT_OQPD <i>reg1, reg2, reg3</i> | VCMPPD <i>reg1, reg2, reg3, 1EH</i> |
| VCMPLT_OQPD <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 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
 8: OP5 \leftarrow EQ_UQ;
 9: OP5 \leftarrow NGE_US;
 10: OP5 \leftarrow NGT_US;
 11: OP5 \leftarrow FALSE_OQ;
 12: OP5 \leftarrow NEQ_OQ;
 13: OP5 \leftarrow GE_OS;
 14: OP5 \leftarrow GT_OS;
 15: OP5 \leftarrow TRUE_UQ;
 16: OP5 \leftarrow EQ_OS;
 17: OP5 \leftarrow LT_OQ;
 18: OP5 \leftarrow LE_OQ;
 19: OP5 \leftarrow UNORD_S;
 20: OP5 \leftarrow NEQ_US;
 21: OP5 \leftarrow NLT_UQ;
 22: OP5 \leftarrow NLE_UQ;
 23: OP5 \leftarrow ORD_S;
 24: OP5 \leftarrow EQ_US;
 25: OP5 \leftarrow NGE_UQ;
 26: OP5 \leftarrow NGT_UQ;
 27: OP5 \leftarrow FALSE_OS;
 28: OP5 \leftarrow NEQ_OS;
 29: OP5 \leftarrow GE_OQ;
 30: OP5 \leftarrow GT_OQ;
 31: OP5 \leftarrow TRUE_US;
 DEFAULT: Reserved;

ESAC;

VCMPDD (VEX.256 encoded version)

CMP0 \leftarrow SRC1[63:0] OP5 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP5 SRC2[127:64];
 CMP2 \leftarrow SRC1[191:128] OP5 SRC2[191:128];
 CMP3 \leftarrow SRC1[255:192] OP5 SRC2[255:192];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 000000000000000H; FI;
 IF CMP2 = TRUE

INSTRUCTION SET REFERENCE

```
    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;
```

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[255:128] ← 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[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCMPD __m256 _mm256_cmp_pd(__m256 a, __m256 b, const int imm)
```

```
VCMPD __m128 _mm_cmp_pd(__m128 a, __m128 b, const int imm)
```

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-9.

Denormal

Other Exceptions

See Exceptions Type 2

CMPPS- Compare Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8 | 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.128.0F C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8 | 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.256.0F C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8 | V/V | AVX | Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate |

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

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 (255:128) of the corresponding YMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand.

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 (255:128) of the destination YMM 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, bits 4:0 define the type of comparison to be performed (see Figure 5-9). 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 5-9). 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 5-12. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-12. 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> |
| CMPLEPS <i>xmm1, xmm2</i> | CMPPS <i>xmm1, xmm2, 2</i> |
| CMPUNORDPS <i>xmm1, xmm2</i> | CMPPS <i>xmm1, xmm2, 3</i> |
| CMPNEQPS <i>xmm1, xmm2</i> | CMPPS <i>xmm1, xmm2, 4</i> |
| CMPNLTPS <i>xmm1, xmm2</i> | CMPPS <i>xmm1, xmm2, 5</i> |
| CMPNLEPS <i>xmm1, xmm2</i> | CMPPS <i>xmm1, xmm2, 6</i> |

Table 5-12. Pseudo-Op and CMPPS Implementation

| Pseudo-Op | CMPPS Implementation |
|-----------------------------|----------------------------|
| CMPOORDPS <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 5-13, 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 5-13, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-13. 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> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 2</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 3</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 4</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 5</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 6</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 7</i> |
| VCMPEQ_UQPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 8</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 9</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 0AH</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 0BH</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 0CH</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 0DH</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 0EH</i> |
| VCMPLTPS <i>reg1, reg2, reg3</i> | VCMPPS <i>reg1, reg2, reg3, 0FH</i> |

Table 5-13. Pseudo-Op and VCMPPS Implementation

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

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
- 1: OP3 ← LT_OS; OP5 ← LT_OS;
- 2: OP3 ← LE_OS; OP5 ← LE_OS;
- 3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
- 4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
- 5: OP3 ← NLT_US; OP5 ← NLT_US;
- 6: OP3 ← NLE_US; OP5 ← NLE_US;
- 7: OP3 ← ORD_Q; OP5 ← ORD_Q;
- 8: OP5 ← EQ_UQ;
- 9: OP5 ← NGE_US;
- 10: OP5 ← NGT_US;
- 11: OP5 ← FALSE_OQ;
- 12: OP5 ← NEQ_OQ;
- 13: OP5 ← GE_OS;
- 14: OP5 ← GT_OS;

15: OP5 \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 (VEX.256 encoded version)

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

INSTRUCTION SET REFERENCE

ELSE DEST[159:128] \leftarrow 000000000H; FI;
IF CMP5 = TRUE
THEN DEST[191:160] \leftarrow FFFFFFFFH;
ELSE DEST[191:160] \leftarrow 000000000H; FI;
IF CMP6 = TRUE
THEN DEST[223:192] \leftarrow FFFFFFFFH;
ELSE DEST[223:192] \leftarrow 000000000H; FI;
IF CMP7 = TRUE
THEN DEST[255:224] \leftarrow FFFFFFFFH;
ELSE DEST[255:224] \leftarrow 000000000H; FI;

VCMPSS (VEX.128 encoded version)

CMP0 \leftarrow SRC1[31:0] OP5 SRC2[31:0];
CMP1 \leftarrow SRC1[63:32] OP5 SRC2[63:32];
CMP2 \leftarrow SRC1[95:64] OP5 SRC2[95:64];
CMP3 \leftarrow SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
THEN DEST[31:0] \leftarrow FFFFFFFFH;
ELSE DEST[31:0] \leftarrow 000000000H; FI;
IF CMP1 = TRUE
THEN DEST[63:32] \leftarrow FFFFFFFFH;
ELSE DEST[63:32] \leftarrow 000000000H; FI;
IF CMP2 = TRUE
THEN DEST[95:64] \leftarrow FFFFFFFFH;
ELSE DEST[95:64] \leftarrow 000000000H; FI;
IF CMP3 = TRUE
THEN DEST[127:96] \leftarrow FFFFFFFFH;
ELSE DEST[127:96] \leftarrow 000000000H; FI;
DEST[255:128] \leftarrow 0

CMPPS (128-bit Legacy SSE version)

CMP0 \leftarrow SRC1[31:0] OP3 SRC2[31:0];
CMP1 \leftarrow SRC1[63:32] OP3 SRC2[63:32];
CMP2 \leftarrow SRC1[95:64] OP3 SRC2[95:64];
CMP3 \leftarrow SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
THEN DEST[31:0] \leftarrow FFFFFFFFH;
ELSE DEST[31:0] \leftarrow 000000000H; FI;
IF CMP1 = TRUE
THEN DEST[63:32] \leftarrow FFFFFFFFH;
ELSE DEST[63:32] \leftarrow 000000000H; FI;
IF CMP2 = TRUE
THEN DEST[95:64] \leftarrow FFFFFFFFH;

```
ELSE DEST[95:64] ← 000000000H; FI;  
IF CMP3 = TRUE  
  THEN DEST[127:96] ← FFFFFFFFH;  
  ELSE DEST[127:96] ← 000000000H; FI;  
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSS __m256 __mm256_cmp_ps(__m256 a, __m256 b, const int imm)

VCMPSS __m128 __mm_cmp_ps(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 5-9.

Denormal

Other Exceptions

See Exceptions Type 2

CMPSD- Compare Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8 | 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 C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8 | V/V | AVX | Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate |

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. The comparison result 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 64-bit memory location. Bits (255:64) of the corresponding YMM destination register remain unchanged.

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 (255:128) of the destination YMM register are zeroed.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 5-9). 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 5-9). 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 5-14. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-14. 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 5-15, 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 5-15, where the notations of reg1, reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-15. Pseudo-Op and VCMPSD Implementation

| Pseudo-Op | VCMPSD Implementation |
|---------------------------------------|-------------------------------------|
| VCMPEQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 0</i> |
| VCMPLTSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 1</i> |
| VCMPLESD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 2</i> |
| VCMPUNORDSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 3</i> |
| VCMPNEQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 4</i> |
| 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> |
| VCMPGTSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 0EH</i> |
| VCMPTRUESD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 0FH</i> |
| VCMPEQ_OSSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 10H</i> |
| VCMPLT_OQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 11H</i> |
| VCMPLT_OQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 12H</i> |
| VCMPUNORD_SSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 13H</i> |
| VCMPNEQ_USSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 14H</i> |
| VCMPNLT_UQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 15H</i> |
| VCMPNLE_UQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 16H</i> |
| VCMPORD_SSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 17H</i> |
| VCMPEQ_USSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 18H</i> |
| VCMPNGE_UQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 19H</i> |
| VCMPNGT_UQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 1AH</i> |
| VCMFALSE_OSSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 1BH</i> |
| VCMPNEQ_OSSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 1CH</i> |
| VCMPEQ_OQSD <i>reg1, reg2, reg3</i> | VCMPSD <i>reg1, reg2, reg3, 1DH</i> |

Table 5-15. Pseudo-Op and VCMPSPD Implementation

| Pseudo-Op | VCMPSPD Implementation |
|---------------------------------------|--------------------------------------|
| 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;

INSTRUCTION SET REFERENCE

DEFAULT: Reserved
ESAC;

CMPSD (128-bit Legacy SSE version)

$CMP0 \leftarrow DEST[63:0] \text{ OP3 } SRC[63:0];$
IF $CMP0 = \text{TRUE}$
THEN $DEST[63:0] \leftarrow \text{FFFFFFFFFFFFFFFFH};$
ELSE $DEST[63:0] \leftarrow \text{0000000000000000H};$ FI;
 $DEST[255:64]$ (Unmodified)

VCMPSD (VEX.128 encoded version)

$CMP0 \leftarrow SRC1[63:0] \text{ OP5 } SRC2[63:0];$
IF $CMP0 = \text{TRUE}$
THEN $DEST[63:0] \leftarrow \text{FFFFFFFFFFFFFFFFH};$
ELSE $DEST[63:0] \leftarrow \text{0000000000000000H};$ FI;
 $DEST[127:64] \leftarrow SRC1[127:64]$
 $DEST[255:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

`VCMPSD __m128 _mm_cmp_sd(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 5-9
Denormal.

Other Exceptions

See Exceptions Type 3

CMPSS- Compare Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8 | 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 C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8 | V/V | AVX | Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 4:0 of imm8 as comparison predicate. |

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. The comparison result 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 32-bit memory location. Bits (255:32) of the corresponding YMM destination register remain unchanged.

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 (255:128) of the destination YMM register are zeroed.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 5-9). 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 5-9). 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 CMPSS instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 5-16. Compiler should treat reserved Imm8 values as illegal syntax.

Table 5-16. 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> |
| CMPLESS <i>xmm1, xmm2</i> | CMPSS <i>xmm1, xmm2, 2</i> |
| CMPUNORDSS <i>xmm1, xmm2</i> | CMPSS <i>xmm1, xmm2, 3</i> |
| CMPNEQSS <i>xmm1, xmm2</i> | CMPSS <i>xmm1, xmm2, 4</i> |
| CMPNLTSS <i>xmm1, xmm2</i> | CMPSS <i>xmm1, xmm2, 5</i> |
| CMPNLESS <i>xmm1, xmm2</i> | CMPSS <i>xmm1, xmm2, 6</i> |
| CMPODSS <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 5-15, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCOMPSS instruction. See Table 5-17, where the notations of reg1, reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 5-17. Pseudo-Op and VCMPS Implementation

| Pseudo-Op | VCMPS 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> |
| VCMPLNORDSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 3</i> |
| VCMPLNEQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 4</i> |
| VCMPLNLTSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 5</i> |
| VCMPLNLESS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 6</i> |
| VCMPLORDSS <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> |
| VCMPLNGESS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 9</i> |
| VCMPLNGTSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0AH</i> |
| VCMPLFALSESS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0BH</i> |
| VCMPLNEQ_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0CH</i> |
| VCMPLGESS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0DH</i> |
| VCMPLGTSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 0EH</i> |
| VCMPLTRUESS <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> |
| VCMPL_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 12H</i> |
| VCMPLNORD_SSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 13H</i> |
| VCMPLNEQ_USSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 14H</i> |
| VCMPLNLT_UQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 15H</i> |
| VCMPLNLE_UQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 16H</i> |
| VCMPLORD_SSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 17H</i> |
| VCMPEQ_USSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 18H</i> |
| VCMPLNGE_UQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 19H</i> |
| VCMPLNGT_UQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1AH</i> |
| VCMPLFALSE_OSSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1BH</i> |
| VCMPLNEQ_OSSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1CH</i> |
| VCMPLGE_OQSS <i>reg1, reg2, reg3</i> | VCMPS <i>reg1, reg2, reg3, 1DH</i> |

Table 5-17. Pseudo-Op and VCOMPSS Implementation

| Pseudo-Op | VCOMPSS Implementation |
|--|--------------------------------------|
| VCMPGT_OQSS <i>reg1, reg2, reg3</i> | VCOMPSS <i>reg1, reg2, reg3, 1EH</i> |
| VCMPTTRUE_USSS <i>reg1, reg2, reg3</i> | VCOMPSS <i>reg1, reg2, reg3, 1FH</i> |

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

Operation

CASE (COMPARISON PREDICATE) OF

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

DEFAULT: Reserved
 ESAC;

CMPSS (128-bit Legacy SSE version)

$CMP0 \leftarrow DEST[31:0] \text{ OP3 } SRC[31:0]$;
 IF $CMP0 = \text{TRUE}$
 THEN $DEST[31:0] \leftarrow \text{FFFFFFFFH}$;
 ELSE $DEST[31:0] \leftarrow \text{00000000H}$; FI;
 $DEST[255:32]$ (Unmodified)

VCMPSS (VEX.128 encoded version)

$CMP0 \leftarrow SRC1[31:0] \text{ OP5 } SRC2[31:0]$;
 IF $CMP0 = \text{TRUE}$
 THEN $DEST[31:0] \leftarrow \text{FFFFFFFFH}$;
 ELSE $DEST[31:0] \leftarrow \text{00000000H}$; FI;
 $DEST[127:32] \leftarrow SRC1[127:32]$
 $DEST[255:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

`VCMPSS __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 5-9, Denormal.

Other Exceptions

See Exceptions Type 3

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 2F /r COMISD xmm1, xmm2/m64 | 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 2F /r VCOMISD xmm1, xmm2/m64 | V/V | AVX | Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly. |

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 (#1) 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.

Note: In VEX-encoded versions, VEX.vvvv is 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 \leftarrow 111;

GREATER_THAN: ZF,PF,CF \leftarrow 000;

LESS_THAN: ZF,PF,CF \leftarrow 001;

EQUAL: ZF,PF,CF \leftarrow 100;

ESAC;

OF, AF, SF \leftarrow 0; }

Intel C/C++ Compiler Intrinsic Equivalent

`int __mm_comieq_sd (__m128d a, __m128d b)`

`int __mm_comilt_sd (__m128d a, __m128d b)`

`int __mm_comile_sd (__m128d a, __m128d b)`

`int __mm_comigt_sd (__m128d a, __m128d b)`

`int __mm_comige_sd (__m128d a, __m128d b)`

`int __mm_comineq_sd (__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv \neq 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 0F 2F /r COMISS xmm1, xmm2/m32 | V/V | SSE | Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |
| VEX.128.0F 2F /r VCOMISS xmm1, xmm2/m32 | V/V | AVX | Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |

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 (#1) 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.

Note: In VEX-encoded versions, VEX.vvvv is 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 \leftarrow 111;

GREATER_THAN: ZF,PF,CF \leftarrow 000;

LESS_THAN: ZF,PF,CF \leftarrow 001;

EQUAL: ZF,PF,CF \leftarrow 100;

ESAC;

OF, AF, SF \leftarrow 0; }

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_comieq_ss (__m128 a, __m128 b)

int _mm_comilt_ss (__m128 a, __m128 b)

int _mm_comile_ss (__m128 a, __m128 b)

int _mm_comigt_ss (__m128 a, __m128 b)

int _mm_comige_ss (__m128 a, __m128 b)

int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv \neq 1111B.

CVTDQ2PD- Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64 | 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 E6 /r VCVTDQ2PD xmm1, xmm2/m64 | 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 E6 /r VCVTDQ2PD ymm1, xmm2/m128 | V/V | AVX | Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1 |

Description

Converts two or four packed signed doubleword integers in the source operand (second operand) to two or four packed double-precision floating-point values in the destination operand (first operand).

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

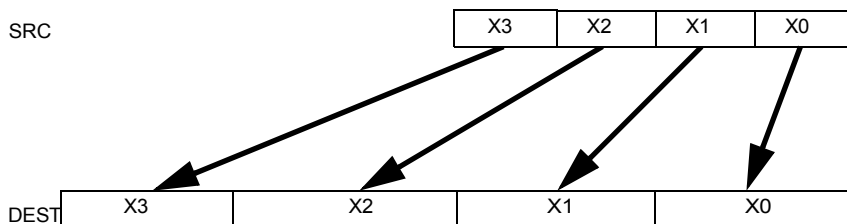


Figure 5-5. CVTDDQ2PD (VEX.256 encoded version)

Operation

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])

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[255:128] ← 0

CVTDDQ2PD (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[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PD __m256d __mm256_cvtepi32_pd (__m128i src)

CVTDDQ2PD __m128d __mm_cvtepi32_pd (__m128i src)

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

CVTDQ2PS- Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

| Opcode Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 5B /r CVTDQ2PS xmm1, xmm2/m128 | V/V | SSE2 | Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1 |
| VEX.128.0F 5B /r VCVTDQ2PS xmm1, xmm2/m128 | V/V | AVX | Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1 |
| VEX.256.0F 5B /r VCVTDQ2PS ymm1, ymm2/m256 | V/V | AVX | Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1 |

Description

Converts four or eight packed signed doubleword integers in the source operand to four or eight packed single-precision floating-point values in the destination operand.

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

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

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] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
 DEST[127:96] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96])
 DEST[159:128] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
 DEST[191:160] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
 DEST[223:192] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
 DEST[255:224] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224])

VCVTDQ2PS (VEX.128 encoded version)

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
 DEST[63:32] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
 DEST[95:64] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
 DEST[127:96] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96])
 DEST[255:128] \leftarrow 0

CVTDQ2PS (128-bit Legacy SSE version)

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
 DEST[63:32] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
 DEST[95:64] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
 DEST[127:96] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96])
 DEST[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PS __m256 __mm256_cvtepi32_ps (__m256i src)

CVTDQ2PS __m128 __mm_cvtepi32_ps (__m128i src)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPD2DQ- Convert Packed Double-Precision Floating-point values to Packed Doubleword Integers

| Opcode Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128 | 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 E6 /r VCVTPD2DQ xmm1, xmm2/m128 | 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 E6 /r VCVTPD2DQ xmm1, ymm2/m256 | V/V | AVX | Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 |

Description

Converts two or four packed double-precision floating-point values in the source operand (second operand) to two or four packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (255:64) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

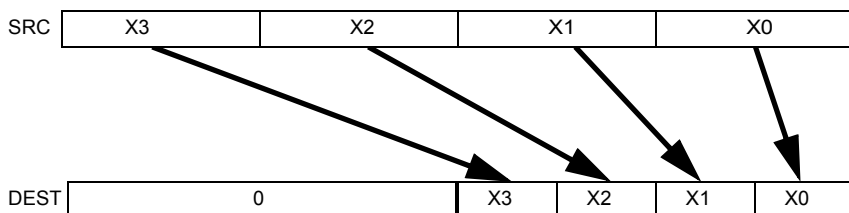


Figure 5-6. VCVTPD2DQ (VEX.256 encoded version)

Operation

VCVTPD2DQ (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[63:0])$
 $\text{DEST}[63:32] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[127:64])$
 $\text{DEST}[95:64] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[191:128])$
 $\text{DEST}[127:96] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[255:192])$
 $\text{DEST}[255:128] \leftarrow 0$

VCVTPD2DQ (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[63:0])$
 $\text{DEST}[63:32] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[127:64])$
 $\text{DEST}[255:64] \leftarrow 0$

CVTPD2DQ (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[63:0])$
 $\text{DEST}[63:32] \leftarrow \text{Convert_Double_Precision_Floating_Point_To_Integer}(\text{SRC}[127:64])$
 $\text{DEST}[127:64] \leftarrow 0$
 $\text{DEST}[255:128]$ (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ __m128i __mm256_cvtpd_epi32 (__m256d src)

CVTPD2DQ __m128i __mm_cvtpd_epi32 (__m128d src)

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPD2PS- Convert Packed Double-Precision Floating-point values to Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 5A /r CVTPD2PS xmm1, xmm2/m128 | 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 5A /r VCVTPD2PS xmm1, xmm2/m128 | 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 5A /r VCVTPD2PS xmm1, ymm2/m256 | V/V | AVX | Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1 |

Description

Converts two or four packed double-precision floating-point values in the source operand (second operand) to two or four 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.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (255:64) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

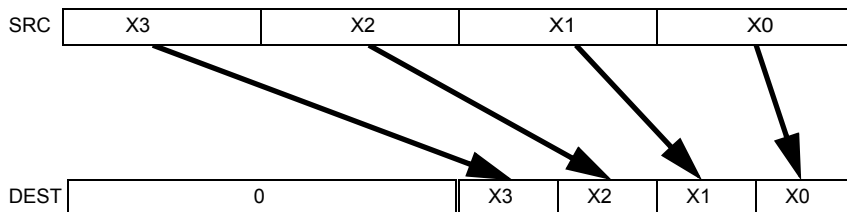


Figure 5-7. VCVTPD2PS (VEX.256 encoded version)

Operation

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[255:128] ← 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[255:64] ← 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[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS __m256 _mm256_cvtpd_ps (__m256d a)

CVTPD2PS __m128 _mm_cvtpd_ps (__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPS2DQ- Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128 | 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 5B /r VCVTPS2DQ xmm1, xmm2/m128 | 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 5B /r VCVTPS2DQ ymm1, ymm2/m256 | V/V | AVX | Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 |

Description

Converts four or eight packed single-precision floating-point values in the source operand to four or eight signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

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[255:128] ← 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[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2DQ `__m256i _mm256_cvtps_epi32 (__m256 a)`

CVTPS2DQ `__m128i _mm_cvtps_epi32 (__m128 a)`

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPS2PD- Convert Packed Single Precision Floating-point values to Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 5A /r CVTPS2PD xmm1, xmm2/m64 | V/V | SSE2 | Convert two packed single-precision floating-point values in xmm2/mem to two packed double-precision floating-point values in xmm1 |
| VEX.128.0F 5A /r VCVTPS2PD xmm1, xmm2/m64 | V/V | AVX | Convert two packed single-precision floating-point values in xmm2/mem to two packed double-precision floating-point values in xmm1 |
| VEX.256.0F 5A /r VCVTPS2PD ymm1, xmm2/m128 | V/V | AVX | Convert four packed single-precision floating-point values in xmm2/mem to four packed double-precision floating-point values in ymm1 |

Description

Converts two or four packed single-precision floating-point values in the source operand (second operand) to two or four packed double-precision floating-point values in the destination operand (first operand).

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

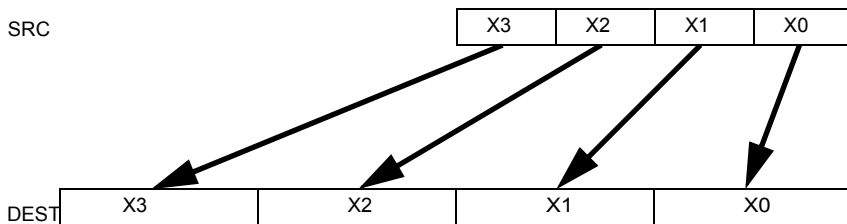


Figure 5-8. CVTTPS2PD (VEX.256 encoded version)

Operation

VCVTTPS2PD (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])

VCVTTPS2PD (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[255:128] ← 0

CVTTPS2PD (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[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2PD __m256d __mm256_cvtps_pd (__m128 a)

CVTTPS2PD __m128d __mm_cvtps_pd (__m128 a)

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTSD2SI- Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F2 0F 2D /r CVTSD2SI r32, xmm1/m64 | V/V | SSE2 | Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32. |
| F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64 | V/N.E. | SSE2 | Convert one double precision floating-point value from xmm/m64 to one signed quadword integer sign-extended into r64. |
| VEX.128.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64 | 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 VCVTSD2SI r64, xmm1/m64 | V/N.E. | AVX | Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64. |

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

INSTRUCTION SET REFERENCE

Note: In VEX-encoded versions, VEX.vvvv is 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

(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

int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTSD2SS- Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64 | 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 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64 | 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. |

Description

Converts a double-precision floating-point value in the second source operand to a single-precision floating-point value in the destination operand.

When the second source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are copied from the upper 3 doublewords of the first source operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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 (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ← SRC1[127:32]
```


INSTRUCTION SET REFERENCE

DEST[255:128] ← 0

CVTSD2SS (128-bit Legacy SSE version)

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[255:32] Unmodified *)

Intel C/C++ Compiler Intrinsic Equivalent

CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

CVTSD- Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F2 0F 2A /r CVTSD xmm1, r32/m32 | 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 CVTSD xmm1, r/m64 | 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 VCVTSD xmm1, xmm2, r/m32 | 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 VCVTSD xmm1, xmm2, r/m64 | V/N.E. | AVX | Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1. |

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

INSTRUCTION SET REFERENCE

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD

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[255:128] ← 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[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SD __m128d_mm_cvtsi32_sd(__m128d a, int b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type 3

CVTSS2SS- Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 2A /r CVTSS2SS xmm1, r/m32 | 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 CVTSS2SS xmm1, r/m64 | 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 VCVTS2SS xmm1, xmm2, r/m32 | 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 VCVTS2SS xmm1, xmm2, r/m64 | V/N.E. | AVX | Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1. |

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255: 32) of the corresponding YMM destination register remain unchanged.

INSTRUCTION SET REFERENCE

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (VEX.128 encoded version)

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);

ELSE

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[127:32] \leftarrow SRC1[127:32]

DEST[255:128] \leftarrow 0

CVTSI2SS (128-bit Legacy SSE version)

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);

ELSE

DEST[31:0] \leftarrow Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SS __m128_mm_cvtsi32_ss(__m128 a, int b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type 3

CVTSS2SD- Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32 | 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 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32 | 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. |

Description

Converts a single-precision floating-point value in the second source operand to a double-precision floating-point value in the destination operand. When the second source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is copied from the first source 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 (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM 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 (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[255:128] ← 0

INSTRUCTION SET REFERENCE

CVTSS2SD (128-bit Legacy SSE version)

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);

DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 b)

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type 3

CVTSS2SI- Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F3 0F 2D /r CVTSS2SI r32, xmm1/m32 | V/V | SSE | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32. |
| F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32 | 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 VCVTSS2SI r32, xmm1/m32 | 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 VCVTSS2SI r64, xmm1/m32 | V/N.E. | AVX | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64. |

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

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

INSTRUCTION SET REFERENCE

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

CVTSS2SI

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

int _mm_cvtss_si32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTTPD2DQ- Convert with Truncation Packed Double-Precision Floating-point values to Packed Doubleword Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128 | 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 E6 /r VCVTPD2DQ xmm1, xmm2/m128 | 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 E6 /r VCVTPD2DQ xmm1, ymm2/m256 | V/V | AVX | Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation |

Description

Converts two or four packed double-precision floating-point values in the source operand (second operand) to two or four packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

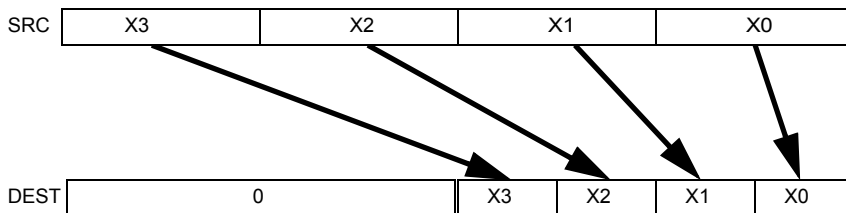


Figure 5-9. VCVTTPD2DQ (VEX.256 encoded version)

Operation

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[255:128] ← 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[255:64] ← 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[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m128i __mm256_cvttpd_epi32 (__m256d src)

CVTTDQ2PD __m128i _mm_cvttpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTTPS2DQ- Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128 | 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 5B /r VCVTTPS2DQ xmm1, xmm2/m128 | 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 5B /r VCVTTPS2DQ ymm1, ymm2/m256 | V/V | AVX | Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation |

Description

Converts four or eight packed single-precision floating-point values in the source operand to four or eight signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

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[255:128] ← 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[255:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 a)

CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTTSD2SI- Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F 2C /r CVTTSD2SI r32, xmm1/m64 | V/V | SSE2 | Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation. |
| F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64 | 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 VCVTTSD2SI r32, xmm1/m64 | 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 VCVTTSD2SI r64, xmm1/m64 | V/N.E. | AVX | Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation. |

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

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

CVTTSD2SI

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

int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTTSS2SI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F3 0F 2C /r CVTTSS2SI r32, xmm1/m32 | V/V | SSE | Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation. |
| F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32 | 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 VCVTTSS2SI r32, xmm1/m32 | 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 VCVTTSS2SI r64, xmm1/m32 | V/N.E. | AVX | Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation. |

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

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

CVTTSS2SI

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);

FI;

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_cvttss_si32(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

DIVPD- Divide Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 5E /r DIVPD xmm1, xmm3/m128 | 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 5E /r VDIVPD xmm1, xmm2, xmm3/m128 | 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 5E /r VDIVPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem |

Description

Performs an SIMD divide of the two or four packed double-precision floating-point values in the first source operand by the two or four packed double-precision floating-point values in the second source operand.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VDIVPD (VEX.256 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] / SRC2[63:0]$

$DEST[127:64] \leftarrow SRC1[127:64] / SRC2[127:64]$

$DEST[191:128] \leftarrow SRC1[191:128] / SRC2[191:128]$

$DEST[255:192] \leftarrow SRC1[255:192] / SRC2[255:192]$

VDIVPD (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] / \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] / \text{SRC2}[127:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$
DIVPD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] / \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] / \text{SRC2}[127:64]$$

$$\text{DEST}[255:128] \text{ (Unmodified)}$$
Intel C/C++ Compiler Intrinsic Equivalent

$$\text{VDIVPD } _m256d _mm256_div_pd (_m256d \text{ a}, _m256d \text{ b});$$

$$\text{DIVPD } _m128d _mm_div_pd (_m128d \text{ a}, _m128d \text{ b});$$
SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

See Exceptions Type 2

DIVPS- Divide Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 0F 5E /r DIVPS xmm1, xmm2/m128 | V/V | SSE | Divide packed single-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem |
| VEX.NDS.128.0F 5E /r VDIVPS xmm1, xmm2, xmm3/m128 | V/V | AVX | Divide packed single-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem |
| VEX.NDS.256.0F 5E /r VDIVPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Divide packed single-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem |

Description

Performs an SIMD divide of the four or eight packed single-precision floating-point values in the first source operand by the four or eight packed single-precision floating-point values in the second source operand.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VDIVPS (VEX.256 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] / SRC2[31:0]$

$DEST[63:32] \leftarrow SRC1[63:32] / SRC2[63:32]$

$DEST[95:64] \leftarrow SRC1[95:64] / SRC2[95:64]$

$DEST[127:96] \leftarrow SRC1[127:96] / SRC2[127:96]$

$\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] / \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] / \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] / \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] / \text{SRC2}[255:224]$.

VDIVPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] / \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] / \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] / \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] / \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

DIVPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] / \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] / \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] / \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] / \text{SRC2}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`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

See Exceptions Type 2

DIVSD- Divide Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F 5E /r DIVSD xmm1, xmm2/m64 | V/V | SSE2 | Divide low double-precision floating point values in xmm1 by low double precision floating-point value in xmm2/mem64. |
| VEX.NDS.128.F2.0F 5E /r VDIVSD xmm1, xmm2, xmm3/m64 | V/V | AVX | Divide low double-precision floating point values in xmm2 by low double precision floating-point value in xmm3/mem64. |

Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination hyperons are XMM registers. The high quadword of the destination operand is copied from the high quadword of the first source operand. See Chapter 11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an overview of a scalar double-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] / \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$

DIVSD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] / \text{SRC}[63:0]$$

DEST[255:64] (Unmodified)

Intel C/C++ 6Compiler Intrinsic Equivalent

DIVSD `__m128d _mm_div_sd (__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

See Exceptions Type 3

DIVSS- Divide Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 5E /r DIVSS xmm1, xmm2/m32 | 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 5E /r VDIVSS xmm1, xmm2, xmm3/m32 | V/V | AVX | Divide low single-precision floating point value in xmm2 by low single precision floating-point value in xmm3/m32. |

Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination are copied from the same dwords of the first source operand. See Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an overview of a scalar single-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] / SRC2[31:0]$

$DEST[127:32] \leftarrow SRC1[127:32]$

$DEST[255:128] \leftarrow 0$

DIVSS (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow DEST[31:0] / SRC[31:0]$

DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

DIVSS __m128 _mm_div_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

See Exceptions Type 3

DPPD- Dot Product of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 3A 41 /r ib DPPD xmm1, xmm3/m128, imm8 | V/V | SSE4_1 | Selectively multiply packed DP floating-point values from xmm1 with packed DP floating-point values from xmm2, add and selectively store the packed DP floating-point values to xmm1 |
| VEX.NDS.128.66.0F3A 41 /r ib VDPPD xmm1,xmm2, xmm3/m128, imm8 | V/V | AVX | Selectively multiply packed DP floating-point values from xmm2 with packed DP floating-point values from xmm3, add and selectively store the packed DP floating-point values to xmm1 |

Description

Conditionally multiplies the packed double precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits 4-5 of the immediate operand. Each of the two resulting double-precision values is summed and this sum is conditionally broadcast to each of 2 positions in the destination operand if the corresponding bit of the mask selected from bits 0-1 of the immediate operand is "1". If the corresponding low bit 0-1 of the mask is zero, the destination is set to zero. DPPD follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

DP_primitive (SRC1, SRC2)

```

IF (imm8[4] == 1) THEN Temp1[63:0] ← SRC1[63:0] * SRC2[63:0];
ELSE Temp1[63:0] ← +0.0;
IF (imm8[5] == 1) THEN Temp1[127:64] ← SRC1[127:64] * SRC2[127:64];
ELSE Temp1[127:64] ← +0.0;
Temp2[63:0] ← Temp1[63:0] + Temp1[127:64];
IF (imm8[0] == 1) THEN DEST[63:0] ← Temp2[63:0];
ELSE DEST[63:0] ← +0.0;
IF (imm8[1] == 1) THEN DEST[127:64] ← Temp2[63:0];
ELSE DEST[127:64] ← +0.0;

```

VDPPD (VEX.128 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← 0

```

DPPD (128-bit Legacy SSE version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```
DPPD __m128d __mm_dp_pd ( __m128d a, __m128d b, const int mask);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched

Other Exceptions

See Exceptions Type 2; additionally

```
#UD                If VEX.L = 1
```

DPPS- Dot Product of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 3A 40 /r ib DPPS xmm1, xmm3/m128, imm8 | V/V | SSE4_1 | Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm3/mem selectively add and store to xmm1 |
| VEX.NDS.128.66.0F3A 40 /r ib VDPPS xmm1,xmm2, xmm3/m128, imm8 | V/V | AVX | Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm2/mem selectively add and store to xmm1 |
| VEX.NDS.256.66.0F3A 40 /r ib VDPPS ymm1, ymm2, ymm3/m256, imm8 | V/V | AVX | Multiply packed single-precision floating-point values from ymm2 with packed SP floating point values from ymm3/mem, selectively add pairs of elements and store to ymm1 |

Description

Multiplies the packed single precision floating point values in the first source operand (second operand) with the packed single-precision floats in the second source (third operand). Each of the four resulting single-precision values is conditionally summed depending on a mask extracted from the high 4 bits of the immediate operand. This sum is broadcast to each of 4 positions in the destination operand (first operand) if the corresponding bit of the mask selected from the low 4 bits of the immediate operand is "1". If the corresponding low bit 0-3 of the mask is zero, the destination is set to zero.

The process is replicated for the high elements of the destination YMM.

DPPS follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] == 1) THEN Temp1[31:0] ← SRC1[31:0] * SRC2[31:0];
    ELSE Temp1[31:0] ← +0.0;
IF (imm8[5] == 1) THEN Temp1[63:32] ← SRC1[63:32] * SRC[63:32];
    ELSE Temp1[63:32] ← +0.0;
IF (imm8[6] == 1) THEN Temp1[95:64] ← SRC1[95:64] * SRC2[95:64];
    ELSE Temp1[95:64] ← +0.0;
IF (imm8[7] == 1) THEN Temp1[127:96] ← SRC1[127:96] * SRC2[127:96];
    ELSE Temp1[127:96] ← +0.0;
```

```
Temp2[31:0] ← Temp1[31:0] + Temp1[63:32];
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96];
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0];
```

```
IF (imm8[0] == 1) THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0;
IF (imm8[1] == 1) THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0;
IF (imm8[2] == 1) THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0;
IF (imm8[3] == 1) THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0;
```

VDPPS (VEX.256 encoded version)

```
DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← DP_Primitive(SRC1[255:128], SRC2[255:128]);
```

VDPPS (VEX.128 encoded version)

```
DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← 0
```

INSTRUCTION SET REFERENCE

DPP (128-bit Legacy SSE version)

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDPPS __m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);

(V)DPPS __m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation.

Unmasked exceptions will leave the destination untouched

Other Exceptions

See Exceptions Type 2

VEEXTRACTF128- Extract packed floating-point values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| VEX.256.66.0F3A 19 /r ib VEEXTRACTF128 xmm1/m128, ymm2, imm8 | V/V | AVX | Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/mem |

Description

Extracts 128-bits of packed floating-point values from the source operand (second operand) at an 128-bit offset from imm8[0] into the destination operand (first operand). The destination may be either an XMM register or a 128-bit memory location.

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

The high 7 bits of the immediate are ignored.

If VEEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

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[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VEEXTRACTF128 __m128 _mm256_extractf128_ps (__m256 a, int offset);

VEEXTRACTF128 __m128d _mm256_extractf128_pd (__m256d a, int offset);

INSTRUCTION SET REFERENCE

VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD IF VEX.L = 0

EXTRACTPS- Extract packed floating-point values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8 | 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 17 /r ib VEXTRACTPS r/m32, xmm1, imm8 | 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. |

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 encoded version: When VEX.128.66.0F3A.W1 17 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits. VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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.

INSTRUCTION SET REFERENCE

Operation

VEXTRACTPS (VEX.128 encoded version)

SRC_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh

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 0FFFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh

FI

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS __mm_extractmem_ps (float *dest, __m128 a, const int nidx);

EXTRACTPS __m128 __mm_extract_ps (__m128 a, const int nidx);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; Additionally

#UD IF VEX.L = 1

HADDPD- Add Horizontal Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 7C /r HADDPD xmm1, xmm2/m128 | V/V | SSE3 | Horizontal add packed double-precision floating-point values from xmm1 and xmm2/mem |
| VEX.NDS.128.66.0F 7C /r VHADDPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Horizontal add packed double-precision floating-point values from xmm2 and xmm3/mem |
| VEX.NDS.256.66.0F 7C /r VHADDPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Horizontal add packed double-precision floating-point values from ymm2 and ymm3/mem |

Description

Adds pairs of adjacent double-precision floating-point values in the first source operand and second source operand and stores results in the destination.

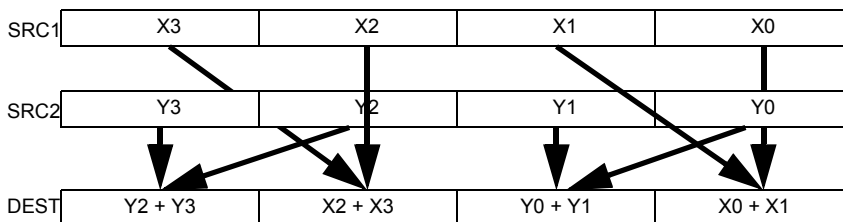


Figure 5-10. VHADDPD operation

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.

INSTRUCTION SET REFERENCE

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHADDPD (VEX.256 encoded version)

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

VHADDPD (VEX.128 encoded version)

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

HADDPD (128-bit Legacy SSE version)

$DEST[63:0] \leftarrow SRC1[127:64] + SRC1[63:0]$
 $DEST[127:64] \leftarrow SRC2[127:64] + SRC2[63:0]$
 $DEST[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD `__m256d _mm256_hadd_pd (__m256d a, __m256d b);`

HADDPD `__m128d _mm_hadd_pd (__m128d a, __m128d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

HADDPS- Add Horizontal Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F2 0F 7C /r HADDPS xmm1, xmm2/m128 | V/V | SSE3 | Horizontal add packed single-precision floating-point values from xmm1 and xmm2/mem |
| VEX.NDS.128.F2.0F 7C /r VHADDPS xmm1, xmm2, xmm3/m128 | V/V | AVX | Horizontal add packed single-precision floating-point values from xmm2 and xmm3/mem |
| VEX.NDS.256.F2.0F 7C /r VHADDPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Horizontal add packed single-precision floating-point values from ymm2 and ymm3/mem |

Description

Adds pairs of adjacent single-precision floating-point values in the first source operand and second source operand and stores results in the destination.

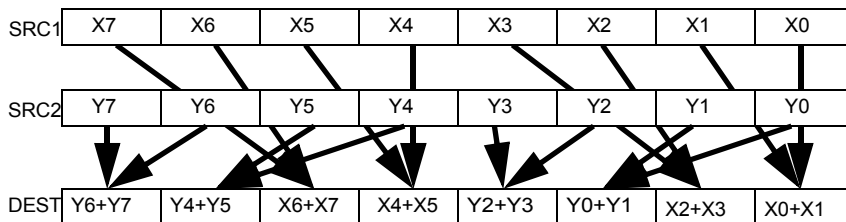


Figure 5-11. VHADDPS operation

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHADDPS (VEX.256 encoded version)

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

VHADDPS (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$
 $DEST[255:128] \leftarrow 0$

HADDPS (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$
 $DEST[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPS `__m256 _mm256_hadd_ps (__m256 a, __m256 b);`

HADDPS `__m128 _mm_hadd_ps (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

HSUBPD- Subtract Horizontal Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 7D /r HSUBPD xmm1, xmm2/m128 | V/V | SSE3 | Horizontal subtract packed double-precision floating-point values from xmm1 and xmm2/mem |
| VEX.NDS.128.66.0F 7D /r VHSUBPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Horizontal subtract packed double-precision floating-point values from xmm2 and xmm3/mem |
| VEX.NDS.256.66.0F 7D /r VHSUBPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Horizontal subtract packed double-precision floating-point values from ymm2 and ymm3/mem |

Description

Subtract pairs of adjacent double-precision floating-point values in the first source operand and second source operand and stores results in the destination.

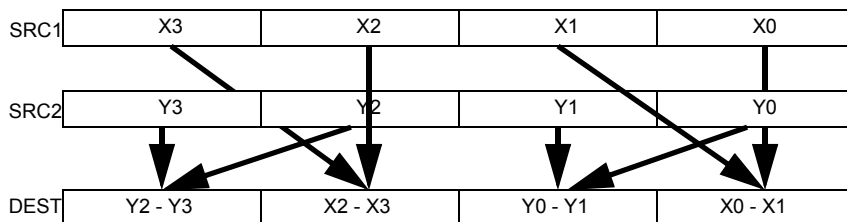


Figure 5-12. VHSUBPD operation

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHSUBPD (VEX.256 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] - SRC1[127:64]$

$DEST[127:64] \leftarrow SRC2[63:0] - SRC2[127:64]$

$DEST[191:128] \leftarrow SRC1[191:128] - SRC1[255:192]$

$DEST[255:192] \leftarrow SRC2[191:128] - SRC2[255:192]$

VHSUBPD (VEX.128 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] - SRC1[127:64]$

$DEST[127:64] \leftarrow SRC2[63:0] - SRC2[127:64]$

$DEST[255:128] \leftarrow 0$

HSUBPD (128-bit Legacy SSE version)

$DEST[63:0] \leftarrow SRC1[63:0] - SRC1[127:64]$

$DEST[127:64] \leftarrow SRC2[63:0] - SRC2[127:64]$

$DEST[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHSUBPD `__m256d _mm256_hsub_pd (__m256d a, __m256d b);`

HSUBPD `__m128d _mm_hsub_pd (__m128d a, __m128d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

HSUBPS- Subtract Horizontal Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F2 0F 7D /r HSUBPS xmm1, xmm2/m128 | V/V | SSE3 | Horizontal subtract packed single-precision floating-point values from xmm1 and xmm2/mem |
| VEX.NDS.128.F2.0F 7D /r VHSUBPS xmm1, xmm2, xmm3/m128 | V/V | AVX | Horizontal subtract packed single-precision floating-point values from xmm2 and xmm3/mem |
| VEX.NDS.256.F2.0F 7D /r VHSUBPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Horizontal subtract packed single-precision floating-point values from ymm2 and ymm3/mem |

Description

Subtract pairs of adjacent single-precision floating-point values in the first source operand and second source operand and stores results in the destination.

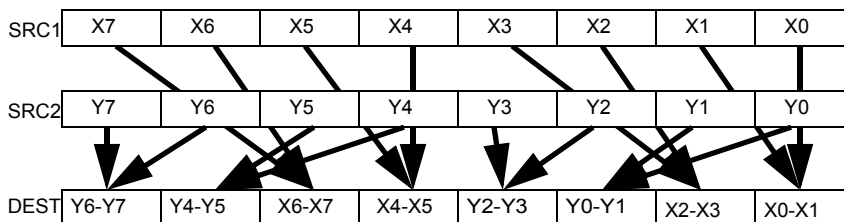


Figure 5-13. VHSUBPS operation

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VHSUBPS (VEX.256 encoded version)

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

VHSUBPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
 DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
 DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
 DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
 DEST[255:128] ← 0

HSUBPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
 DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
 DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
 DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
 DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VHSUBPS __m256 __mm256_hsub_ps (__m256 a, __m256 b);

HSUBPS __m128 __mm_hsub_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 2

VINSERTF128- Insert packed floating-point values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| VEX.NDS.256.66.0F3A 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8 | V/V | AVX | Insert 128-bits of packed floating-point values from xmm3/mem and the remaining values from ymm2 into ymm1 |

Description

Performs an insertion of 128-bits of packed floating-point values from the second source operand (third operand) into an the destination operand (first operand) at an 128-bit offset from imm8[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

Operation

INSERTF128

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

INSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);

INSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);

INSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6

INSERTPS- Insert Scalar Single Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8 | 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 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8 | V/V | AVX | Insert a single precision floating point value selected by imm8 from xmm3/m32 and merge into xmm2 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8. |

Description

(register source form)

Select a single precision floating-point element from second source as indicated by Count_S bits of the immediate operand and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

Load a floating-point element from a 32-bit memory location and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version. The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (255:128) of the corresponding YMM 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 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[255:128] ← 0

```


INSTRUCTION SET REFERENCE

INSERTPS (128-bit Legacy SSE version)

IF (SRC == REG) THEN COUNT_S \leftarrow imm8[7:6]

ELSE COUNT_S \leftarrow 0

COUNT_D \leftarrow imm8[5:4]

ZMASK \leftarrow imm8[3:0]

CASE (COUNT_S) OF

0: TMP \leftarrow SRC[31:0]

1: TMP \leftarrow SRC[63:32]

2: TMP \leftarrow SRC[95:64]

3: TMP \leftarrow SRC[127:96]

ESAC;

CASE (COUNT_D) OF

0: TMP2[31:0] \leftarrow TMP

 TMP2[127:32] \leftarrow DEST[127:32]

1: TMP2[63:32] \leftarrow TMP

 TMP2[31:0] \leftarrow DEST[31:0]

 TMP2[127:64] \leftarrow DEST[127:64]

2: TMP2[95:64] \leftarrow TMP

 TMP2[63:0] \leftarrow DEST[63:0]

 TMP2[127:96] \leftarrow DEST[127:96]

3: TMP2[127:96] \leftarrow TMP

 TMP2[95:0] \leftarrow DEST[95:0]

ESAC;

IF (ZMASK[0] == 1) THEN DEST[31:0] \leftarrow 00000000H

ELSE DEST[31:0] \leftarrow TMP2[31:0]

IF (ZMASK[1] == 1) THEN DEST[63:32] \leftarrow 00000000H

ELSE DEST[63:32] \leftarrow TMP2[63:32]

IF (ZMASK[2] == 1) THEN DEST[95:64] \leftarrow 00000000H

ELSE DEST[95:64] \leftarrow TMP2[95:64]

IF (ZMASK[3] == 1) THEN DEST[127:96] \leftarrow 00000000H

ELSE DEST[127:96] \leftarrow TMP2[127:96]

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

INSETRTPS __m128 __mm_insert_ps(__m128 dst, __m128 src, const int nidx);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

LDDQU- Move Unaligned Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F F0 /r LDDQU xmm1, m128 | V/V | SSE3 | Load unaligned packed integer values from mem to xmm1 |
| VEX.128.F2.0F F0 /r VLDDQU xmm1, m128 | V/V | AVX | Load unaligned packed integer values from mem to xmm1 |
| VEX.256.F2.0F F0 /r VLDDQU ymm1, m256 | V/V | AVX | Load unaligned packed integer values from mem to ymm1 |

Description

The instruction is functionally similar to VMOVDQU YMM, m256 for loading from memory. That is: 32 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32-byte boundary. Up to 64 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to VMOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by VLDDQU be modified and stored to the same location, use VMOVDQU or VMOVDQA instead of VLDDQU. To move double quadwords to or from memory locations that are known to be aligned on 32-byte boundaries, use the VMOVDQA instruction.

Implementation Notes

- If the source is aligned to a 32-byte boundary, based on the implementation, the 32 bytes may be loaded more than once. For that reason, the usage of VLDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using VMOVDQU.
- This instruction is a replacement for VMOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use VMOVDQA store-load pairs when data is 256-bit aligned or VMOVDQU store-load pairs when data is 256-bit unaligned.
- If the memory address is not aligned on 32-byte boundary, some implementations may load up to 64 bytes and return 32 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.

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

Operation

VLDDQU (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VLDDQU (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

LDDQU (128-bit Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU __m128i _mm_lddqu_si128 (__m128i * p);

LDDQU __m256i _mm256_lddqu_si256 (__m256i * p);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies

VLDMXCSR—Load MXCSR Register

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|-----------------------|---------------------------------------|
| VEX.128.0F AE /2 VLDMXCSR <i>m32</i> | V/V | AVX | Load MXCSR register from <i>m32</i> . |

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location.

The VLDMXCSR instruction is typically used in conjunction with the VSTMXCSR instruction for software that use instruction set extensions operating on the YMM state.

The default MXCSR value at reset is 1F80H.

If a VLDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

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

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

Operation

$\text{MXCSR} \leftarrow m32;$

C/C++ Compiler Intrinsic Equivalent

`_mm_setcsr(unsigned int i)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 9; additionally

#GP For an attempt to set reserved bits in MXCSR

MASKMOVDQU- Store Selected Bytes of Double Quadword with NT Hint

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

Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The location of the first byte of the memory location is specified by DI/EDI/RDI and DS registers. The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

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

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the 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 MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.

INSTRUCTION SET REFERENCE

- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

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

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

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

Operation

MASKMOVDQU

IF (MASK[7] = 1)

THEN DEST[DS:DI/EDI/RDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;

IF (MASK[15] = 1)

THEN DEST[DS:DI/EDI/RDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;

(* Repeat operation for 3rd through 14th bytes in source operand *)

IF (MASK[127] = 1)

THEN DEST[DS:DI/EDI/RDI+15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD

If VEX.L= 1.

If VEX.vvvv != 1111B.

VMASKMOV- Conditional SIMD Packed Loads and Stores

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38 2C /r VMASKMOVPS xmm1, xmm2, m128 | V/V | AVX | Conditionally load packed single-precision values from m128 using mask in xmm2 and store in xmm1 |
| VEX.NDS.256.66.0F38 2C /r VMASKMOVPS ymm1, ymm2, m256 | V/V | AVX | Conditionally load packed single-precision values from m256 using mask in ymm2 and store in ymm1 |
| VEX.NDS.128.66.0F38 2D /r VMASKMOVPD xmm1, xmm2, m128 | V/V | AVX | Conditionally load packed double-precision values from m128 using mask in xmm2 and store in xmm1 |
| VEX.NDS.256.66.0F38 2D /r VMASKMOVPD ymm1, ymm2, m256 | V/V | AVX | Conditionally load packed double-precision values from m256 using mask in ymm2 and store in ymm1 |
| VEX.NDS.128.66.0F38 2E /r VMASKMOVPS m128, xmm1, xmm2 | V/V | AVX | Conditionally store packed single-precision values from xmm2 using mask in xmm1 |
| VEX.NDS.256.66.0F38 2E /r VMASKMOVPS m256, ymm1, ymm2 | V/V | AVX | Conditionally store packed single-precision values from ymm2 using mask in ymm1 |
| VEX.NDS.128.66.0F38 2F /r VMASKMOVPD m128, xmm1, xmm2 | V/V | AVX | Conditionally store packed double-precision values from xmm2 using mask in xmm1 |
| VEX.NDS.256.66.0F38 2F /r VMASKMOVPD m256, ymm1, ymm2 | V/V | AVX | Conditionally store packed double-precision values from ymm2 using mask in ymm1 |

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits

associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions

VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VMASKMOVPS -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
```

DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
 DEST[255:128] ← 0

VMASKMOVDPD - 256-bit load

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
 DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
 DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
 DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0

VMASKMOVDPD - 128-bit load

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
 DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
 DEST[255:128] ← 0

VMASKMOVPS - 256-bit store

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

VMASKMOVPS - 128-bit store

IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
 IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
 IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
 IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VMASKMOVDPD - 256-bit store

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

VMASKMOVDPD - 128-bit store

IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
 IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

`__m256 _mm256_maskload_ps(float const *a, __m256i mask)`

INSTRUCTION SET REFERENCE

```
void  _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void  _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
__m128 _mm256_maskload_ps(float const *a, __m128i mask)
void  _mm256_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm256_maskload_pd(double *a, __m128i mask);
void  _mm256_maskstore_pd(double *a, __m128i mask, __m128d b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6

MAXPD- Maximum of Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 5F /r MAXPD xmm1, xmm2/m128 | V/V | SSE2 | Return the maximum double-precision floating-point values between xmm1 and xmm2/mem |
| VEX.NDS.128.66.0F 5F /r VMAXPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the maximum double-precision floating-point values between xmm2 and xmm3/mem |
| VEX.NDS.256.66.0F 5F /r VMAXPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the maximum packed double-precision floating-point values between ymm2 and ymm3/mem |

Description

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

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

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM 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 (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])

```

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[255:128] ← 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[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD `__m256d __mm256_max_pd (__m256d a, __m256d b);`

(V)MAXPD `__m128d __mm_max_pd (__m128d a, __m128d b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 2

MAXPS- Minimum of Packed Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 5F /r MAXPS xmm1, xmm2/m128 | V/V | SSE | Return the maximum single-precision floating-point values between xmm1 and xmm2/mem |
| VEX.NDS.128.0F 5F /r VMAXPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the maximum single-precision floating-point values between xmm2 and xmm3/mem |
| VEX.NDS.256.0F 5F /r VMAXPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the maximum single double-precision floating-point values between ymm2 and ymm3/mem |

Description

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

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

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM 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 (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])

```

VMAXPS (VEX.128 encoded version)

```

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])
DEST[255:128] ← 0

```

MAXPS (128-bit Legacy SSE version)

```

DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[63:32] ← MAX(DEST[63:32], SRC[63:32])
DEST[95:64] ← MAX(DEST[95:64], SRC[95:64])
DEST[127:96] ← MAX(DEST[127:96], SRC[127:96])
DEST[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 2

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F2 0F 5F /r MAXSD xmm1, xmm2/m64 | V/V | SSE2 | Return the maximum scalar double-precision floating-point value between xmm2/mem64 and xmm1. |
| VEX.NDS.128.F2.0F 5F /r VMAXSD xmm1, xmm2, xmm3/m64 | V/V | AVX | Return the maximum scalar double-precision floating-point value between xmm3/mem64 and xmm2. |

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 quad-word of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed. The high quadword of the destination operand is copied from the same bits of first source operand.

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

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

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

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

MAXSD (128-bit Legacy SSE version)

```
DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])
DEST[255:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 5F /r MAXSS xmm1, xmm2/m32 | V/V | SSE | Return the maximum scalar single-precision floating-point value between xmm2/mem32 and xmm1. |
| VEX.NDS.128.F3.0F 5F /r VMAXSS xmm1, xmm2, xmm3/m32 | V/V | AVX | Return the maximum scalar single-precision floating-point value between xmm3/mem32 and xmm2. |

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 double-word 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 (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

```

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[255:128] ← 0

```

MAXSS (128-bit Legacy SSE version)

```

DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[255:32] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```

__m128 _mm_max_ss(__m128 a, __m128 b)

```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

MINPD- Minimum of Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 5D /r MINPD xmm1, xmm2/m128 | V/V | SSE2 | Return the minimum double-precision floating-point values between xmm1 and xmm2/mem |
| VEX.NDS.128.66.0F 5D /r VMINPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the minimum double-precision floating-point values between xmm2 and xmm3/mem |
| VEX.NDS.256.66.0F 5D /r VMINPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem |

Description

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

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

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM 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 (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[255:128] ← 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[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```
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

See Exceptions Type 2

MINPS- Minimum of Packed Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 0F 5D /r MINPS xmm1, xmm2/m128 | V/V | SSE | Return the minimum single-precision floating-point values between xmm1 and xmm2/mem |
| VEX.NDS.128.0F 5D /r VMINPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the minimum single-precision floating-point values between xmm2 and xmm3/mem |
| VEX.NDS.256.0F 5D /r VMINPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem |

Description

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

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

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM 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 (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[255:128] ← 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[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);
```

```
MINPS __m128 _mm_min_ps (__m128 a, __m128 b);
```


INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 2

MINSND- Return Minimum Scalar Double-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F2 0F 5D /r MINSND xmm1, xmm2/m64 | V/V | SSE2 | Return the minimum scalar double precision floating-point value between xmm2/mem64 and xmm1. |
| VEX.NDS.128.F2.0F 5D /r VMINSND xmm1, xmm2, xmm3/m64 | V/V | AVX | Return the minimum scalar double precision floating-point value between xmm3/mem64 and xmm2. |

Description

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

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

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of 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 (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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

INSTRUCTION SET REFERENCE

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 (VEX.128 encoded version)

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← SRC1[127:64]

DEST[255:128] ← 0

MINSD (128-bit Legacy SSE version)

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MINSD __m128d _mm_min_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F3 0F 5D /r MINSS xmm1,xmm2/m32 | V/V | SSE | Return the minimum scalar single precision floating-point value between xmm2/mem32 and xmm1. |
| VEX.NDS.128.F3.0F 5D /r VMINSS xmm1,xmm2, xmm3/m32 | V/V | AVX | Return the minimum scalar single precision floating-point value between xmm3/mem32 and xmm2. |

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 double-word 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 (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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

Operation

MIN(SRC1, SRC2)

INSTRUCTION SET REFERENCE

```
{  
  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;  
}
```

VMINSS (VEX.128 encoded version)

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])

DEST[127:32] ← SRC1[127:32]

DEST[255:128] ← 0

MINSS (128-bit Legacy SSE version)

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

See Exceptions Type 3

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 28 /r MOVAPD xmm1, xmm2/m128 | V/V | SSE2 | Move aligned packed double-precision floating-point values from xmm2/mem to xmm1 |
| 66 0F 29 /r MOVAPD xmm2/m128, xmm1 | V/V | SSE2 | Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem |
| VEX.128.66.0F 28 /r VMOVAPD xmm1, xmm2/m128 | V/V | AVX | Move aligned packed double-precision floating-point values from xmm2/mem to xmm1 |
| VEX.128.66.0F 29 /r VMOVAPD xmm2/m128, xmm1 | V/V | AVX | Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem |
| VEX.256.66.0F 28 /r VMOVAPD ymm1, ymm2/m256 | V/V | AVX | Move aligned packed double-precision floating-point values from ymm2/mem to ymm1 |
| VEX.256.66.0F 29 /r VMOVAPD ymm2/m256, ymm1 | V/V | AVX | Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem |

Description

Moves 2 or 4 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

VEX.256 encoded version:

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

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register destination are zeroed.

Operation

VMOVAPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVAPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

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

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m256d _mm256_load_pd (double const * p);

VMOVAPD _mm256_store_pd(double * p, __m256d a);

MOVAPD __m128d _mm_load_pd (double const * p);

MOVAPD _mm_store_pd(double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE2; additionally

#UD If VEX.vvvv != 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 28 /r MOVAPS xmm1, xmm2/m128 | V/V | SSE | Move aligned packed single-precision floating-point values from xmm2/mem to xmm1 |
| 0F 29 /r MOVAPS xmm2/m128, xmm1 | V/V | SSE | Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem |
| VEX.128.0F 28 /r VMOVAPS xmm1, xmm2/m128 | V/V | AVX | Move aligned packed single-precision floating-point values from xmm2/mem to xmm1 |
| VEX.128.0F 29 /r VMOVAPS xmm2/m128, xmm1 | V/V | AVX | Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem |
| VEX.256.0F 28 /r VMOVAPS ymm1, ymm2/m256 | V/V | AVX | Move aligned packed single-precision floating-point values from ymm2/mem to ymm1 |
| VEX.256.0F 29 /r VMOVAPS ymm2/m256, ymm1 | V/V | AVX | Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem |

Description

Moves 4 or 8 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

VEX.256 encoded version:

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

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVAPS (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVAPS (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

(V)MOVAPS (128-bit store form)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPS __m256 _mm256_load_ps (float const * p);

VMOVAPS _mm256_store_ps(float * p, __m256 a);

MOVAPS __m128 _mm_load_ps (float const * p);

INSTRUCTION SET REFERENCE

MOVAPS _mm_store_ps(float * p, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE; additionally

#UD If VEX.vvvv != 1111B.

MOVD/MOVQ- Move Doubleword and Quadword

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 6E /r MOVD xmm1, r32/m32 | V/V | SSE2 | Move doubleword from r/m32 to xmm1 |
| 66 REX.W 0F 6E /r MOVQ xmm1, r64/m64 | V/N.E. | SSE2 | Move quadword from r/m64 to xmm1 |
| VEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32 | V/V | AVX | Move doubleword from r/m32 to xmm1 |
| VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64 | V/N.E. | AVX | Move quadword from r/m64 to xmm1 |
| 66 0F 7E /r MOVD r32/m32, xmm1 | V/V | SSE2 | Move doubleword from xmm1 register to r/m32 |
| 66 REX.W 0F 7E /r MOVQ r64/m64, xmm1 | V/N.E. | SSE2 | Move quadword from xmm1 register to r/m64 |
| VEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1 | V/V | AVX | Move doubleword from xmm1 register to r/m32 |
| VEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1 | V/N.E. | AVX | Move quadword from xmm1 register to r/m64 |

Description

MOVD/Q with XMM destination:

Moves a dword integer from the source operand and stores it in the low 32-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32-bit register or 32-bit memory location. A REX.W prefix promotes this to copy qword integers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

MOVD/Q with r32/m32 or r64/m64 destination:

INSTRUCTION SET REFERENCE

Stores 32 (64) bits from the low bits of the source XMM register.

Note: In VEX-encoded versions, VEX.vvvv is 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[255:128] (Unmodified)

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

DEST[31:0] ← SRC[31:0]

DEST[255:32] ← 0H

MOVQ (Legacy SSE version when destination is an XMM register)

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← 0H

DEST[255:128] (Unmodified)

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

DEST[63:0] ← SRC[63:0]

DEST[255:64] ← 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

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

See Exceptions Type 5; additionally

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

MOVQ- Move Quadword

| Opcode | Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|-------------|------------------------|--------------------|--|
| F3 0F 7E /r MOVQ xmm1, xmm2/m64 | | V/V | SSE2 | Move quadword from xmm2/m64 to xmm1 |
| VEX.128.F3.0F 7E /r VMOVQ xmm1, xmm2 | | V/V | AVX | Move quadword from xmm2 to xmm1 |
| VEX.128.F3.0F 7E /r VMOVQ xmm1, m64 | | V/V | AVX | Load quadword from m64 to xmm1 |
| 66 0F D6 /r MOVQ xmm1/m64, xmm2 | | V/V | SSE2 | Move quadword from xmm2 register to xmm1/m64 |
| VEX.128.66.0F D6 /r VMOVQ xmm1/m64, xmm2 | | V/V | AVX | Move quadword from xmm2 register to xmm1/m64 |

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: In VEX.128.66.0F D6 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note: In VEX.128.F3.0F 7E version, VEX.vvvv is 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[255:128] (Unmodified)

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

DEST[63:0] ← SRC[63:0]

DEST[255:64] ← 0

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

DEST[63:0] ← SRC[63:0]

DEST[255:64] ← 0

MOVQ (7E) with memory source:

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← 0000000000000000H

DEST[255:128] (Unmodified)

VMOVQ (7E) with memory source:

DEST[63:0] ← SRC[63:0]

DEST[255:64] ← 0000000000000000H

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

MOVQ __m128i_mm_move_epi64(__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

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

MOVDDUP- Replicate Double FP Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F2 0F 12 /r MOVDDUP xmm1, xmm2/m64 | V/V | SSE3 | Move double-precision floating-point values from xmm2/mem and duplicate into xmm1 |
| VEX.128.F2.0F 12 /r VMOVDDUP xmm1, xmm2/m64 | V/V | AVX | Move double-precision floating-point values from xmm2/mem and duplicate into xmm1 |
| VEX.256.F2.0F 12 /r VMOVDDUP ymm1, ymm2/m256 | V/V | AVX | Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1 |

Description

VEX.256 encoded version:

Duplicates even-indexed double-precision floating-point values from the source operand (second operand).

128-bit versions:

Duplicates a single double-precision floating-point value into the destination.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

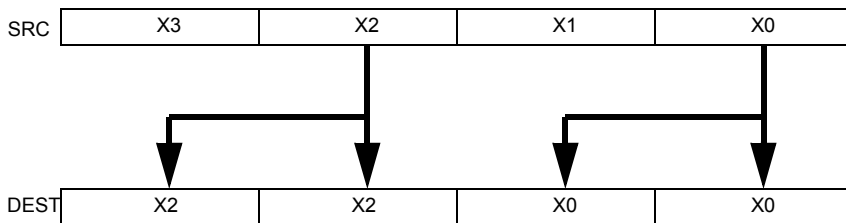


Figure 5-14. VMOVDDUP Operation

Operation

VMOVDDUP (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0]$
 $\text{DEST}[191:128] \leftarrow \text{SRC}[191:128]$
 $\text{DEST}[255:192] \leftarrow \text{SRC}[191:128]$

VMOVDDUP (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0]$
 $\text{DEST}[255:128] \leftarrow 0$

MOVDDUP (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC}[63:0]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`MOVDDUP __m256d _mm256_movedup_pd (__m256d a);`

`MOVDDUP __m128d _mm_movedup_pd (__m128d a);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

MOVDQA- Move Aligned Packed Integer Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 6F /r MOVDQA xmm1, xmm2/m128 | V/V | SSE2 | Move aligned packed integer values from xmm2/mem to xmm1 |
| 66 0F 7F /r MOVDQA xmm2/m128, xmm1 | V/V | SSE2 | Move aligned packed integer values from xmm1 to xmm2/mem |
| VEX.128.66.0F 6F /r VMOVDQA xmm1, xmm2/m128 | V/V | AVX | Move aligned packed integer values from xmm2/mem to xmm1 |
| VEX.128.66.0F 7F /r VMOVDQA xmm2/m128, xmm1 | V/V | AVX | Move aligned packed integer values from xmm1 to xmm2/mem |
| VEX.256.66.0F 6F /r VMOVDQA ymm1, ymm2/m256 | V/V | AVX | Move aligned packed integer values from ymm2/mem to ymm1 |
| VEX.256.66.0F 7F /r VMOVDQA ymm2/m256, ymm1 | V/V | AVX | Move aligned packed integer values from ymm1 to ymm2/mem |

Description

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

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.

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVDQA (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVDQA (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

(V)MOVDQA (128-bit store forms)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA __m256i _mm256_load_si256 (__m256i * p);

VMOVDQA _mm256_store_si256(__m256i *p, __m256i a);

MOVDQA __m128i _mm_load_si128 (__m128i * p);

MOVDQA _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE2; additionally

#UD If VEX.vvvv != 1111B.

MOVDQU- Move Unaligned Packed Integer Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 6F /r MOVDQU xmm1, xmm2/m128 | V/V | SSE2 | Move unaligned packed integer values from xmm2/mem to xmm1 |
| F3 0F 7F /r MOVDQU xmm2/m128, xmm1 | V/V | SSE2 | Move unaligned packed integer values from xmm1 to xmm2/mem |
| VEX.128.F3.0F 6F /r VMOVDQU xmm1, xmm2/m128 | V/V | AVX | Move unaligned packed integer values from xmm2/mem to xmm1 |
| VEX.128.F3.0F 7F /r VMOVDQU xmm2/m128, xmm1 | V/V | AVX | Move unaligned packed integer values from xmm1 to xmm2/mem |
| VEX.256.F3.0F 6F /r VMOVDQU ymm1, ymm2/m256 | V/V | AVX | Move unaligned packed integer values from ymm2/mem to ymm1 |
| VEX.256.F3.0F 7F /r VMOVDQU ymm2/m256, ymm1 | V/V | AVX | Move unaligned packed integer values from ymm1 to ymm2/mem |

Description

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

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.

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 (255:128) of the corresponding YMM destination register remain unchanged.

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

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVDQU (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVDQU (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

(V)MOVDQU 128-bit store-form versions

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQU __m256i _mm256_loadu_si256 (__m256i * p);

VMOVDQU _mm256_storeu_si256(__m256i *p, __m256i a);

MOVDQU __m128i _mm_loadu_si128 (__m128i * p);

MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

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

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

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

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

128-bit three-argument form

Moves two packed single-precision floating-point values from the 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). The upper 128-bits of the destination YMM register are zeroed.

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

Operation

MOVHLPS (128-bit two-argument form)

DEST[63:0] ← SRC[127:64]

DEST[255:64] (Unmodified)

VMOVHLPS (128-bit three-argument form)

DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.L = 1

MOVHPD- Move High Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 16 /r MOVHPD xmm1, m64 | V/V | SSE2 | Move double-precision floating-point values from m64 to high quadword of xmm1. |
| VEX.NDS.128.66.0F 16 /r VMOVHPD xmm2, xmm1, m64 | V/V | AVX | Merge double-precision floating-point value from m64 and the low quadword of xmm1. |
| 66 0F 17/r MOVHPD m64, xmm1 | V/V | SSE2 | Move double-precision floating-point values from high quadword of xmm1 to m64. |
| VEX.128.66.0F 17/r VMOVHPD m64, xmm1 | V/V | AVX | Move double-precision floating-point values from high quadword of xmm1 to m64. |

Description

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

128-bit Legacy SSE load:

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

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

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

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

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

Operation

MOVHPD (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)
 DEST[127:64] \leftarrow SRC[63:0]
 DEST[255:128] (Unmodified)

VMOVHPD (VEX.128 encoded load)

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

VMOVHPD (store)

DEST[63:0] \leftarrow SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally
 #UD If VEX.L = 1.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 16 /r MOVHPS xmm1, m64 | V/V | SSE | Move two packed single-precision floating-point values from m64 to high quadword of xmm1. |
| VEX.NDS.128.0F 16 /r VMOVHPS xmm2, xmm1, m64 | V/V | AVX | Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1. |
| 0F 17/r MOVHPS m64, xmm1 | V/V | SSE | Move two packed single-precision floating-point values from high quadword of xmm1 to m64. |
| VEX.128.0F 17/r VMOVHPS m64, xmm1 | V/V | AVX | Move two packed single-precision floating-point values from high quadword of xmm1 to m64. |

Description

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

128-bit Legacy SSE load:

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

VEX.128 encoded load:

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

128-bit store:

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

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) (VEX.NDS.128.0F 17 /r) instruc-

tion version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

MOVHPS (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)
 DEST[127:64] ← SRC[63:0]
 DEST[255:128] (Unmodified)

VMOVHPS (VEX.128 encoded load)

DEST[63:0] ← SRC1[63:0]
 DEST[127:64] ← SRC2[63:0]
 DEST[255:128] ← 0

VMOVHPS (store)

DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128d _mm_loadh_pi (__m128d a, __m64 *p)
 MOVHPS void _mm_storeh_pi (__m64 *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.

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

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

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

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

128-bit three-argument form

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). The upper 128-bits of the destination YMM register are zeroed.

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

Operation

MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[255:128] (Unmodified)

VMOVLHPS (128-bit three-argument form)

DEST[63:0] ← SRC1[63:0]

DEST[127:64] ← SRC2[63:0]

DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD IF VEX.L = 1.

MOVLPD- Move Low Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 12 /r MOVLPD xmm1, m64 | V/V | SSE2 | Move double-precision floating-point values from m64 to low quadword of xmm1. |
| VEX.NDS.128.66.0F 12 /r VMOVLPD xmm2, xmm1, m64 | V/V | AVX | Merge double-precision floating-point value from m64 and the high quadword of xmm1. |
| 66 0F 13/r MOVLPD m64, xmm1 | V/V | SSE2 | Move double-precision floating-point values from low quadword of xmm1 to m64. |
| VEX.128.66.0F 13/r VMOVLPD m64, xmm1 | V/V | AVX | Move double-precision floating-point values from low quadword of xmm1 to m64. |

Description

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

128-bit Legacy SSE load:

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

VEX.128 encoded load:

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

128-bit store:

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

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

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

Operation

MOVLPD (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]

DEST[255:64] (Unmodified)

VMOVLPD (VEX.128 encoded load)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[255:128] ← 0

VMOVLPD (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD `__m128d _mm_loadl_pd (__m128d a, double *p)`

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

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

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 12 /r MOVLPS xmm1, m64 | V/V | SSE | Move two packed single-precision floating-point values from m64 to low quadword of xmm1. |
| VEX.NDS.128.0F 12 /r VMOVLPS xmm2, xmm1, m64 | V/V | AVX | Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1. |
| 0F 13/r MOVLPS m64, xmm1 | V/V | SSE | Move two packed single-precision floating-point values from low quadword of xmm1 to m64. |
| VEX.128.0F 13/r VMOVLPS m64, xmm1 | V/V | AVX | Move two packed single-precision floating-point values from low quadword of xmm1 to m64. |

Description

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

128-bit Legacy SSE load:

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

VEX.128 encoded load:

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

128-bit store:

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

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

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

Operation

MOVLPS (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]

DEST[255:64] (Unmodified)

VMOVLPS (VEX.128 encoded load)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[255:128] ← 0

VMOVLPS (store)

DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)

MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

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

MOVMSKPD- Extract Double-Precision Floating-Point Sign mask

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 50 /r MOVMSKPD reg, xmm2 | V/V | SSE2 | Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed. |
| VEX.128.66.0F 50 /r VMOVMSKPD reg, xmm2 | V/V | AVX | Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed. |
| VEX.256.66.0F 50 /r VMOVMSKPD reg, ymm2 | V/V | AVX | Extract 4-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zero'ed. |

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2- or 4-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 or 4 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the default operand size of the destination register is 64 bit.

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

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

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

Operation

VMOVMSKPD (VEX.256 encoded version)

DEST[0] ← SRC[63]

DEST[1] ← SRC[127]

DEST[2] ← SRC[191]

DEST[3] ← SRC[255]

IF DEST = r32

THEN DEST[31:4] ← 0;

ELSE DEST[63:4] ← 0;

FI

(V)MOVMSKPD (128-bit versions)

DEST[0] ← SRC[63]

DEST[1] ← SRC[127]

IF DEST = r32

THEN DEST[31:2] ← 0;

ELSE DEST[63:2] ← 0;

FI

Intel C/C++ Compiler Intrinsic Equivalent

int _mm256_movemask_pd(__m256d a)

int _mm_movemask_pd(__m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv != 1111B.

MOVMSKPS- Extract Single-Precision Floating-Point Sign mask

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 50 /r MOVMSKPS reg, xmm2 | V/V | SSE | Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed. |
| VEX.128.0F 50 /r VMOVMSKPS reg, xmm2 | V/V | AVX | Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zero'ed. |
| VEX.256.0F 50 /r VMOVMSKPS reg, ymm2 | V/V | AVX | Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zero'ed. |

Description

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

In 64-bit mode, the default operand size of the destination register is 64 bit.

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

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

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

Operation

VMOVMSKPS (VEX.256 encoded version)

DEST[0] ← SRC[31]
 DEST[1] ← SRC[63]
 DEST[2] ← SRC[95]
 DEST[3] ← SRC[127]
 DEST[4] ← SRC[159]
 DEST[5] ← SRC[191]
 DEST[6] ← SRC[223]

```

DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI

```

(V)MOVMSKPS (128-bit version)

```

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

```

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm256_movemask_ps(__m256 a)
```

```
int _mm_movemask_ps(__m128 a)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv != 1111B.

MOVNTDQ- Store Packed Integers Using Non-Temporal Hint

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F E7 /r MOVNTDQ m128, xmm1 | V/V | SSE2 | Move packed integer values in xmm1 to m128 using non-temporal hint |
| VEX.128.66.0F E7 /r VMOVNTDQ m128, xmm1 | V/V | AVX | Move packed integer values in xmm1 to m128 using non-temporal hint |
| VEX.256.66.0F E7 /r VMOVNTDQ m256, ymm1 | V/V | AVX | Move packed integer values in ymm1 to m256 using non-temporal hint |

Description

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

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an 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: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTDQ

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type1.SSE2; additionally

#UD If VEX.vvvv != 1111B.

MOVNTDQA- Load Double Quadword Non-Temporal Aligned Hint

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 2A /r MOVNTDQA xmm1, m128 | V/V | SSE4_1 | Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type. |
| VEX.128.66.0F38 2A /r VMOVNTDQA xmm1, m128 | V/V | AVX | Move double quadword from m128 to xmm using non-temporal hint if WC memory type. |

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 non-temporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

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

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

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

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTDQA (128bit- Legacy SSE form)

DEST ← SRC

DEST[255:128] (Unmodified)

VMOVNTDQA (VEX.128 encoded form)

DEST ← SRC

DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQA __m128i _mm_stream_load_si128 (__m128i *p);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1.SSE4.1; additionally

#UD If VEX.vvvv != 1111B.

If VEX.L = 1.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 2B /r MOVNTPD m128, xmm1 | V/V | SSE2 | Move packed double-precision values in xmm1 to m128 using non-temporal hint |
| VEX.128.66.0F 2B /r VMOVNTPD m128, xmm1 | V/V | AVX | Move packed double-precision values in xmm1 to m128 using non-temporal hint |
| VEX.256.66.0F 2B /r VMOVNTPD m256, ymm1 | V/V | AVX | Move packed double-precision values in ymm1 to m256 using non-temporal hint |

Description

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

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the 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: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTPD

DEST \leftarrow SRC

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type1.SSE2; additionally

#UD If VEX.vvvv \neq 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 2B /r MOVNTPS m128, xmm1 | V/V | SSE | Move packed single-precision values xmm1 to mem using non-temporal hint |
| VEX.128.0F 2B /r VMOVNTPS m128, xmm1 | V/V | AVX | Move packed single-precision values xmm1 to mem using non-temporal hint |
| VEX.256.0F 2B /r VMOVNTPS m256, ymm1 | V/V | AVX | Move packed single-precision values ymm1 to mem using non-temporal hint |

Description

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

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the 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: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVNTPS

DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type1.SSE; additionally

#UD If VEX.vvvv != 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F 10 /r MOVSD xmm1, xmm2 | V/V | SSE2 | Merge or Move scalar double-precision floating-point value from xmm2 to xmm1 register |
| F2 0F 10 /r MOVSD xmm1, m64 | V/V | SSE2 | Merge or Move scalar double-precision floating-point value from m64 to xmm1 register |
| VEX.NDS.128.F2.0F 10 /r VMOVSD xmm1, xmm2, xmm3 | V/V | AVX | Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register |
| VEX.128.F2.0F 10 /r VMOVSD xmm1, m64 | V/V | AVX | Load scalar double-precision floating-point value from m64 to xmm1 register |
| F2 0F 11 /r MOVSD xmm2/m64, xmm1 | V/V | SSE2 | Move scalar double-precision floating-point value from xmm1 register to xmm2/m64 |
| VEX.NDS.128.F2.0F 11 /r VMOVSD xmm1, xmm2, xmm3 | V/V | AVX | Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 |
| VEX.128.F2.0F 11 /r VMOVSD m64, xmm1 | V/V | AVX | Move scalar double-precision floating-point value from xmm1 register to m64 |

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.

When the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory

location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

Note: For the “VMOVSD m64, xmm1” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b, otherwise instruction will #UD.

Note: For the “VMOVSD xmm1, m64” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

Operation

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)

DEST[63:0] ← SRC[63:0]

DEST[255:64] (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[255:128] ← 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[255:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)

DEST[63:0] ← SRC[63:0]

DEST[255:64] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MOVSD __m128d _mm_load_sd (double *p)

MOVSD void _mm_store_sd (double *p, __m128d a)

MOVSD __m128d _mm_move_sd (__m128d a, __m128d b)

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

MOVSHDUP- Replicate Single FP Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128 | V/V | SSE3 | Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1 |
| VEX.128.F3.0F 16 /r VMOVSHDUP xmm1, xmm2/m128 | V/V | AVX | Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1 |
| VEX.256.F3.0F 16 /r VMOVSHDUP ymm1, ymm2/m256 | V/V | AVX | Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1 |

Description

Duplicates odd-indexed single-precision floating-point values from the source operand (second operand). See Figure 5-15. The source operand is an XMM or YMM register or 128 or 256-bit memory location and the destination operand is an XMM or YMM register.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

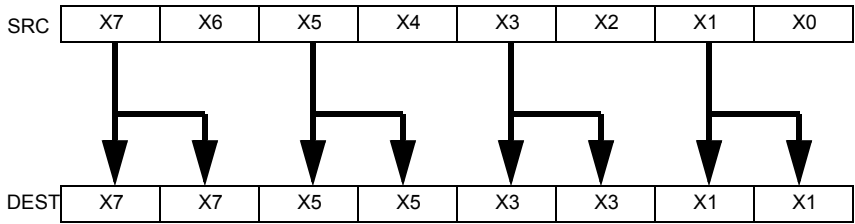


Figure 5-15. MOVSHDUP Operation

Operation

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]

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
 VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 2

MOVSLDUP- Replicate Single FP Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128 | V/V | SSE3 | Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1 |
| VEX.128.F3.0F 12 /r VMOVSLDUP xmm1, xmm2/m128 | V/V | AVX | Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1 |
| VEX.256.F3.0F 12 /r VMOVSLDUP ymm1, ymm2/m256 | V/V | AVX | Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1 |

Description

Duplicates even-indexed single-precision floating-point values from the source operand (second operand). See Figure 5-16. The source operand is an XMM or YMM register or 128 or 256-bit memory location and the destination operand is an XMM or YMM register.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

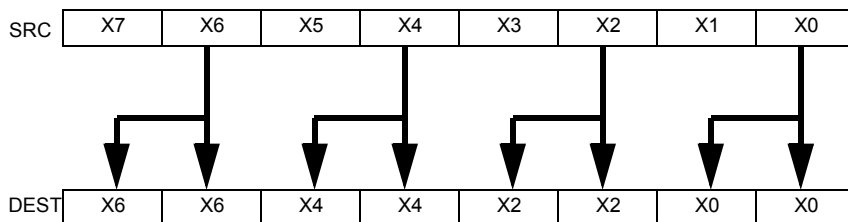


Figure 5-16. MOVSLDUP Operation

Operation

VMOVSLDUP (VEX.256 encoded version)

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

VMOVSLDUP (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC}[95:64]$
 $\text{DEST}[255:128] \leftarrow 0$

MOVSLDUP (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC}[31:0]$
 $\text{DEST}[95:64] \leftarrow \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC}[95:64]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`VMOVSLDUP __m256 __mm256_moveldup_ps (__m256 a);`
`VMOVSLDUP __m128 __mm_moveldup_ps (__m128 a);`

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F3 0F 10 /r MOVSS xmm1, xmm2 | V/V | SSE | Merge scalar single-precision floating-point value from xmm2 to xmm1 register |
| F3 0F 10 /r MOVSS xmm1, m32 | V/V | SSE | Load scalar single-precision floating-point value from m32 to xmm1 register |
| VEX.NDS.128.F3.0F 10 /r VMOVSS xmm1, xmm2, xmm3 | V/V | AVX | Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register |
| VEX.128.F3.0F 10 /r VMOVSS xmm1, m32 | V/V | AVX | Load scalar single-precision floating-point value from m32 to xmm1 register |
| F3 0F 11 /r MOVSS xmm2/m32, xmm1 | V/V | SSE | Move scalar single-precision floating-point value from xmm1 register to xmm2/m32 |
| VEX.NDS.128.F3.0F 11 /r VMOVSS xmm1, xmm2, xmm3 | V/V | AVX | Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register |
| VEX.128.F3.0F 11 /r VMOVSS m32, xmm1 | V/V | AVX | Move scalar single-precision floating-point value from xmm1 register to m32 |

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.

When the source and destination operands are XMM registers, the high doublewords of the destination operand remains unchanged. When the source operand is a

INSTRUCTION SET REFERENCE

memory location and destination operand is an XMM registers, the high doublewords of the destination operand is cleared to all 0s.

Note: For the “VMOVSS m32, xmm1” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the “VMOVSS xmm1, m32” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

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

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

DEST[31:0] ← SRC[31:0]

DEST[255:32] (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[255:128] ← 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[255:128] ← 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[255:32] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MOVSS __m128 _mm_load_ss(float * p)

MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128_mm_move_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 10 /r MOVUPD xmm1, xmm2/m128 | V/V | SSE2 | Move unaligned packed double-precision floating-point from xmm2/mem to xmm1 |
| 66 0F 11 /r MOVUPD xmm2/m128, xmm1 | V/V | SSE2 | Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem |
| VEX.128.66.0F 10 /r VMOVUPD xmm1, xmm2/m128 | V/V | AVX | Move unaligned packed double-precision floating-point from xmm2/mem to xmm1 |
| VEX.128.66.0F 11 /r VMOVUPD xmm2/m128, xmm1 | V/V | AVX | Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem |
| VEX.256.66.0F 10 /r VMOVUPD ymm1, ymm2/m256 | V/V | AVX | Move unaligned packed double-precision floating-point from ymm2/mem to ymm1 |
| VEX.256.66.0F 11 /r VMOVUPD ymm2/m256, ymm1 | V/V | AVX | Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem |

Description

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

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.

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 (255:128) of the corresponding YMM destination register remain unchanged.

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

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVUPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVUPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

(V)MOVUPD (128-bit store form)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPD __m256d _mm256_loadu_pd (__m256d * p);

VMOVUPD _mm256_storeu_pd(__m256d *p, __m256d a);

MOVUPD __m128d _mm_loadu_pd (__m128d * p);

MOVUPD _mm_storeu_pd(__m128d *p, __m128d a);

SIMD Floating-Point Exceptions

None

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD If VEX.vvvv != 1111B.

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

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 10 /r MOVUPS xmm1, xmm2/m128 | V/V | SSE | Move unaligned packed single-precision floating-point from xmm2/mem to xmm1 |
| 0F 11 /r MOVUPS xmm2/m128, xmm1 | V/V | SSE | Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem |
| VEX.128.0F 10 /r VMOVUPS xmm1, xmm2/m128 | V/V | AVX | Move unaligned packed single-precision floating-point from xmm2/mem to xmm1 |
| VEX.128.0F 11 /r VMOVUPS xmm2/m128, xmm1 | V/V | AVX | Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem |
| VEX.256.0F 10 /r VMOVUPS ymm1, ymm2/m256 | V/V | AVX | Move unaligned packed single-precision floating-point from ymm2/mem to ymm1 |
| VEX.256.0F 11 /r VMOVUPS ymm2/m256, ymm1 | V/V | AVX | Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem |

Description

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

VEX.256 encoded version:

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

128-bit versions:

INSTRUCTION SET REFERENCE

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 (255:128) of the corresponding YMM destination register remain unchanged.

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

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

Operation

VMOVUPS (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

VMOVUPS (VEX.128 encoded load-form)

DEST[127:0] ← SRC[127:0]

DEST[255:128] ← 0

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

DEST[127:0] ← SRC[127:0]

DEST[255:128] (Unmodified)

(V)MOVUPS (128-bit store form)

DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVUPS __m256 _mm256_loadu_ps (__m256 * p);
```

```
VMOVUPS _mm256_storeu_ps(_m256 *p, __m256 a);
```

```
MOVUPS __m128 _mm_loadu_ps (__m128 * p);
```

```
MOVUPS _mm_storeu_ps(__m128 *p, __m128 a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally
#UD If VEX.vvvv != 1111B.

MPSADBW - Multiple Sum of Absolute Differences

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 42 /r ib MPSADBW xmm1, xmm2/m128, imm8 | V/V | SSE4_1 | Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm1 and xmm2/m128 and writes the results in xmm1. Starting offsets within xmm1 and xmm2/m128 are determined by imm8 |
| VEX.NDS.128.66.0F3A 42 /r ib VMPSADBW xmm1, xmm2, xmm3/m128, imm8 | V/V | AVX | Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and xmm3/m128 and writes the results in xmm1. Starting offsets within xmm2 and xmm3/m128 are determined by imm8 |

Description

MPSADBW sums the absolute difference of 4 unsigned bytes selected by immediate bits 0-1 from the second source with sequential groups of 4 unsigned bytes in the first source operand. The source bytes from the first source operand start at an offset determined by bit 2 of the immediate. The operation is repeated 8 times, each time using the same second source input but selecting the group of 4 bytes starting at the next higher byte in the first source. Each 16-bit sum is written to dest.

The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: The first source and destination are the same. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

Operation

VMPSADBW (VEX.128 encoded version)

SRC2_OFFSET ← imm8[1:0]*32

SRC1_OFFSET ← imm8[2]*32

SRC1_BYTE0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]

$SRC1_BYTE1 \leftarrow SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]$
 $SRC1_BYTE2 \leftarrow SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]$
 $SRC1_BYTE3 \leftarrow SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]$
 $SRC1_BYTE4 \leftarrow SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]$
 $SRC1_BYTE5 \leftarrow SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]$
 $SRC1_BYTE6 \leftarrow SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]$
 $SRC1_BYTE7 \leftarrow SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]$
 $SRC1_BYTE8 \leftarrow SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]$
 $SRC1_BYTE9 \leftarrow SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]$
 $SRC1_BYTE10 \leftarrow SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]$

$SRC2_BYTE0 \leftarrow SRC2[SRC2_OFFSET+7:SRC2_OFFSET]$
 $SRC2_BYTE1 \leftarrow SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]$
 $SRC2_BYTE2 \leftarrow SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]$
 $SRC2_BYTE3 \leftarrow SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]$

$TEMP0 \leftarrow ABS(SRC1_BYTE0 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE3)$
 $DEST[15:0] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(SRC1_BYTE1 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE3)$
 $DEST[31:16] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(SRC1_BYTE2 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE3)$
 $DEST[47:32] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(SRC1_BYTE3 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE3)$
 $DEST[63:48] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(SRC1_BYTE4 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE1)$
 $TEMP2 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE2)$
 $TEMP3 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE3)$
 $DEST[79:64] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(SRC1_BYTE5 - SRC2_BYTE0)$
 $TEMP1 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE1)$

INSTRUCTION SET REFERENCE

TEMP2 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 \leftarrow ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 \leftarrow ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 \leftarrow ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 \leftarrow ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 \leftarrow ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3

DEST[255:128] \leftarrow 0

MPSADBW (128-bit Legacy SSE version)

SRC_OFFSET \leftarrow imm8[1:0]*32
DEST_OFFSET \leftarrow imm8[2]*32
DEST_BYTE0 \leftarrow DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 \leftarrow DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 \leftarrow DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 \leftarrow DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 \leftarrow DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 \leftarrow DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 \leftarrow DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 \leftarrow DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 \leftarrow DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 \leftarrow DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 \leftarrow DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 \leftarrow SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYTE1 \leftarrow SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYTE2 \leftarrow SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYTE3 \leftarrow SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMP0 \leftarrow ABS(DEST_BYTE0 - SRC_BYTE0)
TEMP1 \leftarrow ABS(DEST_BYTE1 - SRC_BYTE1)
TEMP2 \leftarrow ABS(DEST_BYTE2 - SRC_BYTE2)
TEMP3 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 \leftarrow ABS(DEST_BYTE1 - SRC_BYTE0)

$TEMP1 \leftarrow ABS(DEST_BYTE2 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE3)$
 $DEST[31:16] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(DEST_BYTE2 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE3)$
 $DEST[47:32] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(DEST_BYTE3 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE3)$
 $DEST[63:48] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(DEST_BYTE4 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE3)$
 $DEST[79:64] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(DEST_BYTE5 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE8 - SRC_BYTE3)$
 $DEST[95:80] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $TEMP0 \leftarrow ABS(DEST_BYTE6 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE8 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE9 - SRC_BYTE3)$
 $DEST[111:96] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(DEST_BYTE7 - SRC_BYTE0)$
 $TEMP1 \leftarrow ABS(DEST_BYTE8 - SRC_BYTE1)$
 $TEMP2 \leftarrow ABS(DEST_BYTE9 - SRC_BYTE2)$
 $TEMP3 \leftarrow ABS(DEST_BYTE10 - SRC_BYTE3)$
 $DEST[127:112] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$
 $DEST[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MPSADBW __m128i __mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);

SIMD Floating-Point Exceptions

None

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1

MULPD- Multiply Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 59 /r MULPD xmm1, xmm2/m128 | V/V | SSE2 | Multiply packed double-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1 |
| VEX.NDS.128.66.0F 59 /r VMULPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Multiply packed double-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1 |
| VEX.NDS.256.66.0F 59 /r VMULPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Multiply packed double-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1 |

Description

Performs a SIMD Multiply of the two or four 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.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

INSTRUCTION SET REFERENCE

.

VMULPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] * \text{SRC2}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] * \text{SRC2}[127:64]$

$\text{DEST}[255:128] \leftarrow 0$

MULPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] * \text{SRC}[63:0]$

$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] * \text{SRC}[127:64]$

$\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`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

See Exceptions Type 2

MULPS- Multiply Packed Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 59 /r MULPS xmm1, xmm2/m128 | V/V | SSE | Multiply packed single-precision floating-point values from xmm2/mem to xmm1 and stores result in xmm1 |
| VEX.NDS.128.0F 59 /r VMULPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Multiply packed single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1 |
| VEX.NDS.256.0F 59 /r VMULPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Multiply packed single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1 |

Description

Performs an SIMD multiply of the four or eight packed single-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.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VMULPS (VEX.256 encoded version)

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$$

$$\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$$

$$\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$$

$$\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$$

INSTRUCTION SET REFERENCE

$\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] * \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] * \text{SRC2}[255:224]$.

VMULPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

MULPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`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

See Exceptions Type 2

MULSD- Multiply Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| F2 0F 59 /r MULSD xmm1,xmm2/m64 | V/V | SSE2 | Multiply the low double-precision floating-point value in xmm2/mem64 by low double precision floating-point value in xmm1. |
| VEX.NDS.128.F2.0F 59/r VMULSD xmm1,xmm2, xmm3/m64 | V/V | AVX | Multiply the low double-precision floating-point value in xmm3/mem64 by low double precision floating-point value in xmm2. |

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. The high quadword of the destination operand is copied from the high bits of the first source operand. See Figure 11-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar double-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

$DEST[63:0] \leftarrow SRC1[63:0] * SRC2[63:0]$

$DEST[127:64] \leftarrow SRC1[127:64]$

$DEST[255:128] \leftarrow 0$

INSTRUCTION SET REFERENCE

MULSD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] * \text{SRC}[63:0]$

DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

MULSS- Multiply Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F3 0F 59 /r MULSS xmm1,xmm2/m32 | V/V | SSE | Multiply the low single-precision floating-point value in xmm2/mem by the low single-precision floating-point value in xmm1. |
| VEX.NDS.128.F3.0F 59 /r VMULSS xmm1,xmm2, xmm3/m32 | V/V | AVX | Multiply the low single-precision floating-point value in xmm3/mem by the low single-precision floating-point value in xmm2. |

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. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] * SRC2[31:0]$

$DEST[127:32] \leftarrow SRC1[127:32]$

$DEST[255:128] \leftarrow 0$

INSTRUCTION SET REFERENCE

MULSS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0]$

$\text{DEST}[255:32]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

ORPD- Bitwise Logical OR of Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 56/r ORPD xmm1, xmm2/m128 | 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 VORPD xmm1,xmm2, xmm3/m128 | 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 VORPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) 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 (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

VORPD (VEX.256 encoded version)

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

INSTRUCTION SET REFERENCE

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]

.

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

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

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

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) 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 (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

VORPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

INSTRUCTION SET REFERENCE

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].

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

See Exceptions Type 4

PABSB/PABSW/PABSD - Packed Absolute Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 1C /r PABSB xmm1, xmm2/m128 | V/V | SSSE3 | Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1. |
| 66 0F 38 1D /r PABSW xmm1, xmm2/m128 | 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 PABSD xmm1, xmm2/m128 | V/V | SSSE3 | Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1. |
| VEX.128.66.0F38 1C /r VPABSB xmm1, xmm2/m128 | V/V | AVX | Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1. |
| VEX.128.66.0F38 1D /r VPABSW xmm1, xmm2/m128 | V/V | AVX | Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1. |
| VEX.128.66.0F38 1E /r VPABSD xmm1, xmm2/m128 | V/V | AVX | Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1. |

Description

PABSB/W/D computes the absolute value of each data element of the source operand and stores the UNSIGNED results in the destination operand. PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers. The source is an XMM register or a 128-bit memory location. The destination operand is an XMM register.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation**BYTE_ABS(SRC)**

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

WORD_ABS(SRC)

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

DWORD_ABS(SRC)

```
{
  DEST [31:0] ← ABS(SRC[31:0])
  DEST [63:32] ← ABS(SRC[63:32])
  DEST [95:64] ← ABS(SRC[95:64])
  DEST [127..96] ← ABS(SRC[127:96])
}
```

VPABS (VEX.128 encoded version)

```
DEST[127:0] ← BYTE_ABS(SRC)
DEST[255:128] ← 0
```

PABS (128-bit Legacy SSE version)

```
DEST[127:0] ← BYTE_ABS(SRC)
DEST[255:128] (Unmodified)
```

VPABSW (VEX.128 encoded version)

```
DEST[127:0] ← WORD_ABS(SRC)
DEST[255:128] ← 0
```

PABSW (128-bit Legacy SSE version)

```
DEST[127:0] ← WORD_ABS(SRC)
DEST[255:128] (Unmodified)
```

VPABSD (VEX.128 encoded version)

```
DEST[127:0] ← DWORD_ABS(SRC)
DEST[255:128] ← 0
```

PABSD (128-bit Legacy SSE version)

DEST[127:0] ← DWORD_ABS(SRC)

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PABSB __m128i _mm_abs_epi8 (__m128i a)

PABSW __m128i _mm_abs_epi16 (__m128i a)

PABSD __m128i _mm_abs_epi32 (__m128i a)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

| | |
|-----|-----------------------|
| #UD | If VEX.L = 1. |
| | If VEX.vvvv != 1111B. |

PACKSSWB/PACKSSDW- Pack with Signed Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 63 /r PACKSSWB xmm1,xmm2/m128 | V/V | SSE2 | Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed signed byte integers in xmm1 using signed saturation. |
| 66 0F 6B /r PACKSSDW xmm1,xmm2/m128 | V/V | SSE2 | Converts 4 packed signed doubleword integers from xmm1 and from xmm2/m128 into 8 packed signed word integers in xmm1 using signed saturation. |
| VEX.NDS.128.66.0F 63 /r VPACKSSWB xmm1,xmm2, xmm3/m128 | V/V | AVX | Converts 8 packed signed word integers from xmm2 and from xmm3/m128 into 16 packed signed byte integers in xmm1 using signed saturation. |
| VEX.NDS.128.66.0F 6B /r VPACKSSDW xmm1,xmm2, xmm3/m128 | V/V | AVX | Converts 4 packed signed doubleword integers from xmm2 and from xmm3/m128 into 8 packed signed word integers in xmm1 using signed saturation. |

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 5-17 for an example of the packing operation.

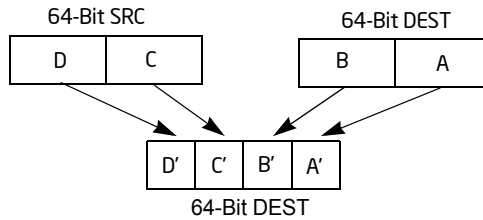


Figure 5-17. PACKSSDW Instruction Operation using 64-bit Operands

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

The PACKSSDW instruction packs 4 signed doublewords from the first source operand and 4 signed doublewords from the second source operand into 8 signed words in the destination operand (see Figure 5-17).

If a signed doubleword integer value is beyond the range of a signed word (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed word integer value of 7FFFH or 8000H, respectively, is stored into the destination.

When operating on 128-bit operands, the first source and destination operands are XMM registers, and the second source operand can be either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

SATURATING_PACK_WB(SRC1, SRC2)

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])

INSTRUCTION SET REFERENCE

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])

SATURATING_PACK_DW(SRC1, SRC2)

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])

PACKSSDW

DEST[127:0] ← SATURATING_PACK_DW(DEST, SRC)
DEST[255:128] (Unmodified)

VPACKSSDW

DEST[127:0] ← SATURATING_PACK_DW(SRC1, SRC2)
DEST[255:128] ← 0

PACKSSWB

DEST[127:0] ← SATURATING_PACK_WB(DEST, SRC)
DEST[255:128] (Unmodified)

VPACKSSWB

DEST[127:0] ← SATURATING_PACK_WB(SRC1, SRC2)
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)

PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PACKUSWB/PACKUSDW- Pack with Unsigned Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 67 /r PACKUSWB xmm1,xmm2/m128 | V/V | SSE2 | Converts 8 signed word integers from xmm1 and 8 signed word integers from xmm2/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation. |
| 66 0F 38 2B /r PACKUSDW xmm1, xmm2/m128 | V/V | SSE4_1 | Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation. |
| VEX.NDS.128.66.0F 67 /r VPACKUSWB xmm1, xmm2, xmm3/m128 | V/V | AVX | Converts 8 signed word integers from xmm2 and 8 signed word integers from xmm3/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation. |
| VEX.NDS.128.66.0F38 2B /r VPACKUSDW xmm1, xmm2, xmm3/m128 | V/V | AVX | Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation. |

Description

packuswb:

Converts 8 signed word integers from the second source operand and 8 signed word integers from the first source operand into 8 unsigned byte integers and stores the result in the destination operand. (See Figure 5-17 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

The first source operand and destination operand must be an XMM register and the second source operand can be either an XMM register or a 128-bit memory location.

packusdw:

Converts packed signed doubleword integers into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively stored in the destination.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

SaturateSignedWordToUnsignedByte(SRC)

```
{
    TMP ← SRC < 0 ? 0 : SRC
    return SRC > FFH ? FFH: TMP
}
```

SaturateSignedDWordToUnsignedWord(SRC)

```
{
    TMP ← SRC < 0 ? 0 : SRC
    return SRC > FFFFH ? FFFFH: TMP
}
```

UNSIGNED_SATURATING_PACK_DW(SRC1, SRC2)

```
DEST[15:0] ← SaturateSignedDWordToUnsignedWord(SRC1[31:0])
DEST[31:16] ← SaturateSignedDWordToUnsignedWord(SRC1[63:32])
DEST[47:32] ← SaturateSignedDWordToUnsignedWord(SRC1[95:64])
DEST[63:48] ← SaturateSignedDWordToUnsignedWord(SRC1[127:96])
DEST[79:64] ← SaturateSignedDWordToUnsignedWord(SRC2[31:0])
DEST[95:80] ← SaturateSignedDWordToUnsignedWord(SRC2[63:32])
DEST[111:96] ← SaturateSignedDWordToUnsignedWord(SRC2[95:64])
DEST[127:112] ← SaturateSignedDWordToUnsignedWord(SRC2[127:96])
```

UNSIGNED_SATURATING_PACK_WB(SRC1, SRC2)

```
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])
```

INSTRUCTION SET REFERENCE

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])

VPACKUSWB (VEX.128 encoded version)

DEST[127:0] ← UNSIGNED_SATURATING_PACK_WB(SRC1, SRC2)
DEST[255:128] ← 0

VPACKUSDW (VEX.128 encoded version)

DEST[127:0] ← UNSIGNED_SATURATING_PACK_DW(SRC1, SRC2)
DEST[255:128] ← 0

PACKUSWB (128-bit Legacy SSE version)

DEST[127:0] ← UNSIGNED_SATURATING_PACK_WB(DEST, SRC)
DEST[255:128] (Unmodified)

PACKUSDW (128-bit Legacy SSE version)

DEST[127:0] ← UNSIGNED_SATURATING_PACK_DW(DEST, SRC)
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PACKUSDW __m128i _mm_packus_epi32(__m128i m1, __m128i m2);

PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDB/PADDW/PADDD/PADDQ- Add Packed Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F FC /r PADDB xmm1, xmm2/m128 | V/V | SSE2 | Add packed byte integers from xmm2/m128 and xmm1. |
| 66 0F FD /r PADDW xmm1, xmm2/m128 | V/V | SSE2 | Add packed word integers from xmm2/m128 and xmm1. |
| 66 0F FE /r PADDD xmm1, xmm2/m128 | V/V | SSE2 | Add packed doubleword integers from xmm2/m128 and xmm1. |
| 66 0F D4 /r PADDQ xmm1, xmm2/m128 | V/V | SSE2 | Add packed quadword integers from xmm2/m128 and xmm1. |
| VEX.NDS.128.66.0F FC /r VPADDB xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed byte integers from xmm3/m128 and xmm2. |
| VEX.NDS.128.66.0F FD /r VPADDW xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed word integers from xmm3/m128 and xmm2. |
| VEX.NDS.128.66.0F FE /r VPADDD xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed doubleword integers from xmm3/m128 and xmm2. |
| VEX.NDS.128.66.0F D4 /r VPADDQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed quadword integers from xmm3/m128 and xmm2. |

Description

Adds the packed byte, word, doubleword, or quadword integers in the first source operand to the second source operand and stores the result in the destination operand. The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operand are XMM registers. 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 of the values operated on.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPADDB (VEX.128 encoded version)

$DEST[7:0] \leftarrow SRC1[7:0] + SRC2[7:0]$
 $DEST[15:8] \leftarrow SRC1[15:8] + SRC2[15:8]$
 $DEST[23:16] \leftarrow SRC1[23:16] + SRC2[23:16]$
 $DEST[31:24] \leftarrow SRC1[31:24] + SRC2[31:24]$
 $DEST[39:32] \leftarrow SRC1[39:32] + SRC2[39:32]$
 $DEST[47:40] \leftarrow SRC1[47:40] + SRC2[47:40]$
 $DEST[55:48] \leftarrow SRC1[55:48] + SRC2[55:48]$
 $DEST[63:56] \leftarrow SRC1[63:56] + SRC2[63:56]$
 $DEST[71:64] \leftarrow SRC1[71:64] + SRC2[71:64]$
 $DEST[79:72] \leftarrow SRC1[79:72] + SRC2[79:72]$
 $DEST[87:80] \leftarrow SRC1[87:80] + SRC2[87:80]$
 $DEST[95:88] \leftarrow SRC1[95:88] + SRC2[95:88]$
 $DEST[103:96] \leftarrow SRC1[103:96] + SRC2[103:96]$
 $DEST[111:104] \leftarrow SRC1[111:104] + SRC2[111:104]$
 $DEST[119:112] \leftarrow SRC1[119:112] + SRC2[119:112]$
 $DEST[127:120] \leftarrow SRC1[127:120] + SRC2[127:120]$
 $DEST[255:128] \leftarrow 0$

PADDB (128-bit Legacy SSE version)

$DEST[7:0] \leftarrow DEST[7:0] + SRC[7:0]$
 $DEST[15:8] \leftarrow DEST[15:8] + SRC[15:8]$
 $DEST[23:16] \leftarrow DEST[23:16] + SRC[23:16]$
 $DEST[31:24] \leftarrow DEST[31:24] + SRC[31:24]$
 $DEST[39:32] \leftarrow DEST[39:32] + SRC[39:32]$
 $DEST[47:40] \leftarrow DEST[47:40] + SRC[47:40]$
 $DEST[55:48] \leftarrow DEST[55:48] + SRC[55:48]$
 $DEST[63:56] \leftarrow DEST[63:56] + SRC[63:56]$
 $DEST[71:64] \leftarrow DEST[71:64] + SRC[71:64]$
 $DEST[79:72] \leftarrow DEST[79:72] + SRC[79:72]$
 $DEST[87:80] \leftarrow DEST[87:80] + SRC[87:80]$
 $DEST[95:88] \leftarrow DEST[95:88] + SRC[95:88]$
 $DEST[103:96] \leftarrow DEST[103:96] + SRC[103:96]$

$\text{DEST}[111:104] \leftarrow \text{DEST}[111:104] + \text{SRC}[111:104]$
 $\text{DEST}[119:112] \leftarrow \text{DEST}[119:112] + \text{SRC}[119:112]$
 $\text{DEST}[127:120] \leftarrow \text{DEST}[127:120] + \text{SRC}[127:120]$
 $\text{DEST}[255:128]$ (Unmodified)

VPADDW (VEX.128 encoded version)

$\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] + \text{SRC2}[15:0]$
 $\text{DEST}[31:16] \leftarrow \text{SRC1}[31:16] + \text{SRC2}[31:16]$
 $\text{DEST}[47:32] \leftarrow \text{SRC1}[47:32] + \text{SRC2}[47:32]$
 $\text{DEST}[63:48] \leftarrow \text{SRC1}[63:48] + \text{SRC2}[63:48]$
 $\text{DEST}[79:64] \leftarrow \text{SRC1}[79:64] + \text{SRC2}[79:64]$
 $\text{DEST}[95:80] \leftarrow \text{SRC1}[95:80] + \text{SRC2}[95:80]$
 $\text{DEST}[111:96] \leftarrow \text{SRC1}[111:96] + \text{SRC2}[111:96]$
 $\text{DEST}[127:112] \leftarrow \text{SRC1}[127:112] + \text{SRC2}[127:112]$
 $\text{DEST}[255:128] \leftarrow 0$

PADDW (128-bit Legacy SSE version)

$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0] + \text{SRC}[15:0]$
 $\text{DEST}[31:16] \leftarrow \text{DEST}[31:16] + \text{SRC}[31:16]$
 $\text{DEST}[47:32] \leftarrow \text{DEST}[47:32] + \text{SRC}[47:32]$
 $\text{DEST}[63:48] \leftarrow \text{DEST}[63:48] + \text{SRC}[63:48]$
 $\text{DEST}[79:64] \leftarrow \text{DEST}[79:64] + \text{SRC}[79:64]$
 $\text{DEST}[95:80] \leftarrow \text{DEST}[95:80] + \text{SRC}[95:80]$
 $\text{DEST}[111:96] \leftarrow \text{DEST}[111:96] + \text{SRC}[111:96]$
 $\text{DEST}[127:112] \leftarrow \text{DEST}[127:112] + \text{SRC}[127:112]$
 $\text{DEST}[255:128]$ (Unmodified)

VPADD (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

PADD (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] + \text{SRC}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{DEST}[63:32] + \text{SRC}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{DEST}[95:64] + \text{SRC}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{DEST}[127:96] + \text{SRC}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

VPADDQ (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$

INSTRUCTION SET REFERENCE

DEST[127:64] ← SRC1[127:64]+SRC2[127:64]
DEST[255:128] ← 0

PADDQ (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0]+SRC[63:0]
DEST[127:64] ← DEST[127:64]+SRC[127:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PADDB __m128i_mm_add_epi8 (__m128ia, __m128ib)
PADDW __m128i_mm_add_epi16 (__m128i a, __m128i b)
PADDD __m128i_mm_add_epi32 (__m128i a, __m128i b)
PADDQ __m128i_mm_add_epi64 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDSB/PADDSW- Add Packed Signed Integers with Signed Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F EC /r PADDSB xmm1, xmm2/m128 | V/V | SSE2 | Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results. |
| 66 0F ED /r PADDSW xmm1, xmm2/m128 | V/V | SSE2 | Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results. |
| VEX.NDS.128.66.0F EC /r VPADDSB xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results. |
| VEX.NDS.128.66.0F ED /r VPADDSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed signed word integers from xmm3/m128 and xmm2 and saturate the results. |

Description

Performs a SIMD add of the packed signed integers from the second source operand and the first source operand and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation.

Overflow is handled with signed saturation, as described in the following paragraphs. The second source operand can be either an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

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

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

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

INSTRUCTION SET REFERENCE

Operation

VPADDSB

DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[255:128] ← 0

PADDSB

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
DEST[255:128] (Unmodified)

VPADDSW

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[255:128] ← 0

PADDSW

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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)

PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PADDUSB/PADDUSW- Add Packed Unsigned Integers with Unsigned Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F DC /r PADDUSB xmm1, xmm2/m128 | V/V | SSE2 | Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results. |
| 66 0F DD /r PADDUSW xmm1, xmm2/m128 | V/V | SSE2 | Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results. |
| VEX.NDS.128.660F DC /r VPADDUSB xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed unsigned byte integers from xmm3/m128 and xmm2 saturate the results. |
| VEX.NDS.128.66.0F DD /r VPADDUSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results. |

Description

Performs a SIMD add of the packed unsigned integers from the second source operand and the first source operand and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

The first source operand and the destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location.

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

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

INSTRUCTION SET REFERENCE

Operation

VPADDUSB

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[255:128] ← 0

PADDUSB

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[111:120] + SRC[127:120]);
DEST[255:128] (Unmodified)

VPADDUSW

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[255:128] ← 0

PADDUSW

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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PADDUSB __m128i _mm_adds_epu8 (__m128i a, __m128i b)

PADDUSW __m128i _mm_adds_epu16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PALIGNR - Byte Align

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 0F /r ib PALIGNR xmm1, xmm2/m128, imm8 | 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 VPALIGNR xmm1, xmm2, xmm3/m128, imm8 | V/V | AVX | Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1 |

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 first source and destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. The immediate value is considered unsigned. Immediate shift counts larger than 32 for 128-bit operands produces a zero result.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PALIGNR

temp1[255:0] ← CONCATENATE(DEST, SRC) >> (imm8*8)

DEST[127:0] ← temp1[127:0]

DEST[255:128] (Unmodified)

VPALIGNR

temp1[255:0] ← CONCATENATE(SRC1, SRC2) >> (imm8*8)

DEST[127:0] ← temp1[127:0]

DEST[255:128] ← 0

INSTRUCTION SET REFERENCE

Intel C/C++ Compiler Intrinsic Equivalent

PALIGNR __m128i __mm_alignr_epi8 (__m128i a, __m128i b, int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PAND- Logical AND

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|------------------------------------|
| 66 0F DB /r PAND xmm1, xmm2/m128 | V/V | SSE2 | Bitwise AND of xmm2/m128 and xmm1. |
| VEX.NDS.128.66.0F DB /r VPAND xmm1, xmm2, xmm3/m128 | V/V | AVX | Bitwise AND of xmm2/m128 and xmm1. |

Description

Performs a bitwise logical AND operation on the second source operand and the first source operand and stores the result in the destination operand. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. 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.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPAND (VEX.128 encoded version)

DEST ← SRC1 AND SRC2

DEST[255:128] ← 0

PAND (128-bit Legacy SSE version)

DEST ← DEST AND SRC

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PAND __m128i __mm_and_si128 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PANDN- Logical AND NOT

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F DF /r PANDN xmm1, xmm2/m128 | V/V | SSE2 | Bitwise AND NOT of xmm2/m128 and xmm1. |
| VEX.NDS.128.66.0F DF /r VPANDN xmm1, xmm2, xmm3/m128 | V/V | AVX | Bitwise AND NOT of xmm2/m128 and xmm1. |

Description

Performs a bitwise logical NOT operation on the first source operand and then performs a bitwise logical AND with the second source operand and stores the result in the destination operand. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. 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.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPANDN (VEX.128 encoded version)

$$\text{DEST} \leftarrow \text{NOT}(\text{SRC1}) \text{ AND } \text{SRC2}$$

$$\text{DEST}[255:128] \leftarrow 0$$

PANDN(128-bit Legacy SSE version)

$$\text{DEST} \leftarrow \text{NOT}(\text{DEST}) \text{ AND } \text{SRC}$$

$$\text{DEST}[255:128] \text{ (Unmodified)}$$

Intel C/C++ Compiler Intrinsic Equivalent

$$\text{PANDN } _m128i _mm_andnot_si128 (_m128i \text{ a}, _m128i \text{ b})$$

SIMD Floating-Point Exceptions

none

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PAVGB/PAVGW - Average Packed Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F E0 /r PAVGB xmm1, xmm2/m128 | V/V | SSE2 | Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding. |
| 66 0F E3 /r PAVGW xmm1, xmm2/m128 | V/V | SSE2 | Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding. |
| VEX.NDS.128.66.0F E0 /r VPAVGB xmm1, xmm2, xmm3/m128 | V/V | AVX | Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding. |
| VEX.NDS.128.66.0F E3 /r VPAVGW xmm1, xmm2, xmm3/m128 | V/V | AVX | Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding. |

Description

Performs a SIMD average of the packed unsigned integers from the second source operand and the first source operand and stores the results in the destination operand. For each corresponding pair of data elements in the first and second source operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The destination and first source operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPAVGB (VEX.128 encoded version)

$$\text{DEST}[7:0] \leftarrow (\text{SRC1}[7:0] + \text{SRC2}[7:0] + 1) \gg 1;$$

(* Repeat operation performed for bytes 2 through 15 *)

$$\text{DEST}[127:120] \leftarrow (\text{SRC1}[127:120] + \text{SRC2}[127:120] + 1) \gg 1$$

$$\text{DEST}[255:128] \leftarrow 0$$

INSTRUCTION SET REFERENCE

PAVGB (128-bit Legacy SSE version)

$DEST[7:0] \leftarrow (SRC[7:0] + DEST[7:0] + 1) \gg 1;$
(* Repeat operation performed for bytes 2 through 15 *)
 $DEST[127:120] \leftarrow (SRC[127:120] + DEST[127:120] + 1) \gg 1$
 $DEST[255:128]$ (Unmodified)

VPAVGW (VEX.128 encoded version)

$DEST[15:0] \leftarrow (SRC1[15:0] + SRC2[15:0] + 1) \gg 1;$
(* Repeat operation performed for 16-bit words 2 through 7 *)
 $DEST[127:112] \leftarrow (SRC1[127:112] + SRC2[127:112] + 1) \gg 1$
 $DEST[255:128] \leftarrow 0$

PAVGW (128-bit Legacy SSE version)

$DEST[15:0] \leftarrow (SRC[15:0] + DEST[15:0] + 1) \gg 1;$
(* Repeat operation performed for 16-bit words 2 through 7 *)
 $DEST[127:112] \leftarrow (SRC[127:112] + DEST[127:112] + 1) \gg 1$
 $DEST[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`PAVGB __m128i _mm_avg_epu8 (__m128i a, __m128i b)`

`PAVGW __m128i _mm_avg_epu16 (__m128i a, __m128i b)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PBLENDVB - Variable Blend Packed Bytes

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 38 10 /r PBLENDVB xmm1, xmm2/m128, <XMM0> | V/V | SSE4_1 | Select byte values from xmm1 and xmm2/m128 using mask bits in the implicit mask register, XMM0, and store the values into xmm1 |
| VEX.NDS.128.66.0F3A 4C /r /is4 VPBLENDVB xmm1, xmm2, xmm3/m128, xmm4 | V/V | AVX | Select byte values from xmm2 and xmm3/m128 using mask bits in the specified mask register, xmm4, and store the values into xmm1 |

Description

Conditionally copy byte elements from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each byte element of the mask register.

Each byte element of the destination operand is copied from:

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

The register assignment of the implicit third operand is defined to be the architectural register XMM0

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (255:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (255:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

VPBLENDVB (VEX.128 encoded version)

MASK \leftarrow SRC3

IF (MASK[7] == 1) THEN DEST[7:0] \leftarrow SRC2[7:0];

ELSE DEST[7:0] \leftarrow SRC1[7:0];

IF (MASK[15] == 1) THEN DEST[15:8] \leftarrow SRC2[15:8];

ELSE DEST[15:8] \leftarrow SRC1[15:8];

IF (MASK[23] == 1) THEN DEST[23:16] \leftarrow SRC2[23:16]

ELSE DEST[23:16] \leftarrow SRC1[23:16];

IF (MASK[31] == 1) THEN DEST[31:24] \leftarrow SRC2[31:24]

ELSE DEST[31:24] \leftarrow SRC1[31:24];

IF (MASK[39] == 1) THEN DEST[39:32] \leftarrow SRC2[39:32]

ELSE DEST[39:32] \leftarrow SRC1[39:32];

IF (MASK[47] == 1) THEN DEST[47:40] \leftarrow SRC2[47:40]

ELSE DEST[47:40] \leftarrow SRC1[47:40];

IF (MASK[55] == 1) THEN DEST[55:48] \leftarrow SRC2[55:48]

ELSE DEST[55:48] \leftarrow SRC1[55:48];

IF (MASK[63] == 1) THEN DEST[63:56] \leftarrow SRC2[63:56]

ELSE DEST[63:56] \leftarrow SRC1[63:56];

IF (MASK[71] == 1) THEN DEST[71:64] \leftarrow SRC2[71:64]

ELSE DEST[71:64] \leftarrow SRC1[71:64];

IF (MASK[79] == 1) THEN DEST[79:72] \leftarrow SRC2[79:72]

ELSE DEST[79:72] \leftarrow SRC1[79:72];

IF (MASK[87] == 1) THEN DEST[87:80] \leftarrow SRC2[87:80]

ELSE DEST[87:80] \leftarrow SRC1[87:80];

IF (MASK[95] == 1) THEN DEST[95:88] \leftarrow SRC2[95:88]

ELSE DEST[95:88] \leftarrow SRC1[95:88];

IF (MASK[103] == 1) THEN DEST[103:96] \leftarrow SRC2[103:96]

ELSE DEST[103:96] \leftarrow SRC1[103:96];

IF (MASK[111] == 1) THEN DEST[111:104] \leftarrow SRC2[111:104]

ELSE DEST[111:104] \leftarrow SRC1[111:104];

IF (MASK[119] == 1) THEN DEST[119:112] \leftarrow SRC2[119:112]

ELSE DEST[119:112] \leftarrow SRC1[119:112];

IF (MASK[127] == 1) THEN DEST[127:120] \leftarrow SRC2[127:120]

ELSE DEST[127:120] \leftarrow SRC1[127:120];

DEST[255:128] \leftarrow 0

PBLENDVB (128-bit Legacy SSE version)

MASK \leftarrow XMM0

IF (MASK[7] == 1) THEN DEST[7:0] \leftarrow SRC[7:0];

ELSE DEST[7:0] \leftarrow DEST[7:0];

IF (MASK[15] == 1) THEN DEST[15:8] \leftarrow SRC[15:8];

ELSE DEST[15:8] \leftarrow DEST[15:8];

```

IF (MASK[23] == 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120]

```

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PBLENDVB __m128i __mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.
 If VEX.W = 1.

PBLENDW - Blend Packed Words

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 0E /r ib PBLENDW xmm1, xmm2/m128, imm8 | V/V | SSE4_1 | Select words from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1 |
| VEX.NDS.128.66.0F3A 0E /r ib VPBLENDW xmm1, xmm2, xmm3/m128, imm8 | V/V | AVX | Select words from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1 |

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPBLENDW (VEX.128 encoded version)

```

IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]

```

```

IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
DEST[255:128] ← 0

```

PBLENDW (128-bit Legacy SSE version)

```

IF (imm8[0] == 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC[127:112]
ELSE DEST[127:112] ← DEST[127:112]

```

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PBLENDW `__m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PCLMULQDQ - Carry-Less Multiplication Quadword

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 3A 44 /r ib PCLMULQDQ <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i> | V/V | CLMUL | Carry-less multiplication of one quadword of <i>xmm1</i> by one quadword of <i>xmm2/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm1</i> and <i>xmm2/m128</i> should be used |

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 5-18, other bits of the immediate byte are ignored.

Table 5-18. PCLMULQDQ Quadword Selection of Immediate Byte

| Imm[4] | Imm[0] | PCLMULQDQ Operation |
|--------|--------|--|
| 0 | 0 | CL_MUL(SRC2 ¹ [63:0], SRC1[63:0]) |
| 0 | 1 | CL_MUL(SRC2[63:0], SRC1[127:64]) |
| 1 | 0 | CL_MUL(SRC2[127:64], SRC1[63:0]) |
| 1 | 1 | CL_MUL(SRC2[127:64], SRC1[127:64]) |

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (255: 128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for *Imm8*.

| Pseudo-Op | Imm8 Encoding |
|--|---------------|
| PCLMULLQLQDQ <i>xmm1</i> , <i>xmm2</i> | 0000_0000B |

| Pseudo-Op | Imm8 Encoding |
|---|---------------|
| PCLMULHQLQDQ <i>xmm1</i> , <i>xmm2</i> | 0000_0001B |
| PCLMULLQHQQDQ <i>xmm1</i> , <i>xmm2</i> | 0001_0000B |
| PCLMULHQQDQ <i>xmm1</i> , <i>xmm2</i> | 0001_0001B |

Operation

PCLMULQDQ

IF (Imm8[0] = 0)

THEN

TEMP1 ← SRC1 [63:0];

ELSE

TEMP1 ← SRC1 [127:64];

FI

IF (Imm8[4] = 0)

THEN

TEMP2 ← SRC2 [63:0];

ELSE

TEMP2 ← SRC2 [127:64];

FI

For i = 0 to 63 {

 TmpB [i] ← (TEMP1[0] and TEMP2[i]);

 For j = 1 to i {

 TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[i - j])

 }

 DEST[i] ← TmpB[i];

}

For i = 64 to 126 {

 TmpB [i] ← 0;

 For j = i - 63 to 63 {

 TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[i - j])

 }

 DEST[i] ← TmpB[i];

}

DEST[127] ← 0;

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PCLMULQDQ `__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)`

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PCMPESTRI - Packed Compare Explicit Length Strings, Return Index

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 61 /r ib PCMPESTRI xmm1, xmm2/m128, imm8 | V/V | SSE4_2 | Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX |
| VEX.128.66.0F3A 61 /r ib VPCMPESTRI xmm1, xmm2/m128, imm8 | V/V | AVX | Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX |

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating an index stored to ECX. Each string is represented by two values. The first value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). The second value is stored in EAX (for xmm1) or EDX (for xmm2/m128) and represents the number of Bytes/Words which are valid for the respective xmm/m128 data. The length of each input is interpreted as being the absolute-value of the value in EAX (EDX). The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in EAX (EDX) is greater than 16 (8) or less than -16 (-8).

At this point the comparisons and aggregation described in section *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed and the index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information.

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise

SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise

OFlag – IntRes2[0]

AFlag – Reset

PFlag – Reset

INSTRUCTION SET REFERENCE

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

See *PCMPESTRI Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);
```

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

```
int _mm_cmpestria (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestric (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestrio (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestris (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestriz (__m128i a, int la, __m128i b, int lb, const int mode);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

| | |
|-----|-----------------------|
| #UD | If VEX.L = 1. |
| | If VEX.vvvv != 1111B. |

PCMPESTRM - Packed Compare Explicit Length Strings, Return Mask

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 3A 60 /r ib PCMPESTRM xmm1, xmm2/m128, imm8 | V/V | SSE4_2 | Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0 |
| VEX.128.66.0F3A 60 /r ib VPCMPESTRM xmm1, xmm2/m128, imm8 | V/V | AVX | Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0 |

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating a mask stored to XMM0. Each string is represented by two values. The first value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). The second value is stored in EAX (for xmm1) or EDX (for xmm2/m128) and represents the number of Bytes/Words which are valid for the respective xmm/m128 data. The length of each input is interpreted as being the absolute-value of the value in EAX (EDX). The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in EAX (EDX) is greater than 16 (8) or less than -16 (-8).

At this point the comparisons and aggregation described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed. As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise

SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise

OFlag –IntRes2[0]

AFlag – Reset

PFlag – Reset

INSTRUCTION SET REFERENCE

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 1, otherwise the instruction will #UD.

Operation

See *PCMPSTRM Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

```
__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);
```

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

```
int _mm_cmpestrma (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestrmc (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestrmo (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestrms (__m128i a, int la, __m128i b, int lb, const int mode);
```

```
int _mm_cmpestrmz (__m128i a, int la, __m128i b, int lb, const int mode);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

| | |
|-----|-----------------------|
| #UD | If VEX.L = 1. |
| | If VEX.vvvv != 1111B. |

PCMPISTRI - Packed Compare Implicit Length Strings, Return Index

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 63 /r ib PCMPISTRI xmm1, xmm2/m128, imm8 | V/V | SSE4_2 | Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX |
| VEX.128.66.0F3A 63 /r ib VPCMPISTRI xmm1, xmm2/m128, imm8 | V/V | AVX | Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX |

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating an index stored to ECX. Each string is represented by a single value. The value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.) At this point the comparisons and aggregation described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed and the index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information.

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise

SFlag – Set if any byte/word of xmm1 is null, reset otherwise

OFlag – IntRes2[0]

AFlag – Reset

PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

INSTRUCTION SET REFERENCE

Operation

See *PCMPISTRI Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

```
int __mm_cmpistri (__m128i a, __m128i b, const int mode);
```

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

```
int __mm_cmpistria (__m128i a, __m128i b, const int mode);
```

```
int __mm_cmpistric (__m128i a, __m128i b, const int mode);
```

```
int __mm_cmpistrio (__m128i a, __m128i b, const int mode);
```

```
int __mm_cmpistris (__m128i a, __m128i b, const int mode);
```

```
int __mm_cmpistriz (__m128i a, __m128i b, const int mode);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

| | |
|-----|-----------------------|
| #UD | If VEX.L = 1. |
| | If VEX.vvvv != 1111B. |

PCMPISTRM - Packed Compare Implicit Length Strings, Return Mask

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 3A 62 /r ib PCMPISTRM xmm1, xmm2/m128, imm8 | V/V | SSE4_2 | Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0 |
| VEX.128.66.0F3A 62 /r ib VPCMPISTRM xmm1, xmm2/m128, imm8 | V/V | AVX | Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0 |

Description

The instruction compares data from two strings based on the control encoded in the imm8 byte (as described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) generating a mask stored to XMM0. Each string is represented by a single value. The value is an XMM (or possibly m128 for the second operand) which contains the elements of the string (character data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

At this point the comparisons and aggregation described in *Section 3.1.2 of Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A* are performed. As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner to supply the most relevant information.

CFlag – Reset if IntRes2 is equal to zero, set otherwise

ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise

SFlag – Set if any byte/word of xmm1 is null, reset otherwise

OFlag – IntRes2[0]

AFlag – Reset

PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

INSTRUCTION SET REFERENCE

Operation

See *PCMPSTRM Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B*.

Intel C/C++ Compiler Intrinsic Equivalent

```
__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode)
```

Intel C/C++ Compiler Intrinsic Equivalent for reading EFLAG Results

```
int _mm_cmpistrma (__m128i a, __m128i b, const int mode);
```

```
int _mm_cmpistrmc (__m128i a, __m128i b, const int mode);
```

```
int _mm_cmpistrmo (__m128i a, __m128i b, const int mode);
```

```
int _mm_cmpistrms (__m128i a, __m128i b, const int mode);
```

```
int _mm_cmpistrmz (__m128i a, __m128i b, const int mode);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

| | |
|-----|-----------------------|
| #UD | If VEX.L = 1. |
| | If VEX.vvvv != 1111B. |

PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ- Compare Packed Integers for Equality

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 74 /r PCMPEQB xmm1, xmm2/m128 | V/V | SSE2 | Compare packed bytes in xmm2/m128 and xmm1 for equality. |
| 66 0F 75 /r PCMPEQW xmm1, xmm2/m128 | V/V | SSE2 | Compare packed words in xmm2/m128 and xmm1 for equality. |
| 66 0F 76 /r PCMPEQD xmm1, xmm2/m128 | V/V | SSE2 | Compare packed doublewords in xmm2/m128 and xmm1 for equality . |
| 66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed quadwords in xmm2/m128 and xmm1 for equality. |
| VEX.NDS.128.66.0F 74 /r VPCMPEQB xmm1, xmm2, xmm3 /m128 | V/V | AVX | Compare packed bytes in xmm3/m128 and xmm2 for equality. |
| VEX.NDS.128.66.0F 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed words in xmm3/m128 and xmm2 for equality. |
| VEX.NDS.128.66.0F 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed doublewords in xmm3/m128 and xmm2 for equality. |
| VEX.NDS.128.66.0F38 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed quadwords in xmm3/m128 and xmm2 for equality. |

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 second source operand can be an XMM

register or a 128-bit memory location. The first source and destination operands are XMM registers.

The PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the PCMPEQW instruction compares the corresponding words in the destination and source operands; the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands, and the PCMPEQQ instruction compares the corresponding quadwords in the destination and source operands.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

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 (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1, SRC2)
DEST[255:128] ← 0

```

PCMPEQB (128-bit Legacy SSE version)

```

DEST[127:0] ← COMPARE_BYTES_EQUAL(DEST, SRC)
DEST[255:128] (Unmodified)

```

VPCMPEQW (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_WORDS_EQUAL(SRC1, SRC2)
DEST[255:128] ← 0

```

PCMPEQW (128-bit Legacy SSE version)

```

DEST[127:0] ← COMPARE_WORDS_EQUAL(DEST, SRC)
DEST[255:128] (Unmodified)

```

VPCMPEQD (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1, SRC2)
DEST[255:128] ← 0

```

PCMPEQD (128-bit Legacy SSE version)

```

DEST[127:0] ← COMPARE_DWORDS_EQUAL(DEST, SRC)
DEST[255:128] (Unmodified)

```

VPCMPEQQ (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1, SRC2)
DEST[255:128] ← 0

```

PCMPEQQ (128-bit Legacy SSE version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(DEST, SRC)
DEST[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

```
PCMPEQB __m128i _mm_cmpeq_epi8 ( __m128i a, __m128i b)
```

```
PCMPEQW __m128i _mm_cmpeq_epi16 ( __m128i a, __m128i b)
```

INSTRUCTION SET REFERENCE

PCMPEQD __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)

PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ- Compare Packed Integers for Greater Than

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 64 /r PCMPGTB xmm1, xmm2/m128 | V/V | SSE2 | Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than. |
| 66 0F 65 /r PCMPGTW xmm1, xmm2/m128 | V/V | SSE2 | Compare packed signed word integers in xmm1 and xmm2/m128 for greater than. |
| 66 0F 66 /r PCMPGTD xmm1, xmm2/m128 | V/V | SSE2 | Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than. |
| 66 0F 38 37 /r PCMPGTQ xmm1, xmm2/m128 | V/V | SSE4_2 | Compare packed qwords in xmm2/m128 and xmm1 for greater than. |
| VEX.NDS.128.66.0F 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than. |
| VEX.NDS.128.66.0F 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed word integers in xmm2 and xmm3/m128 for greater than. |
| VEX.NDS.128.66.0F 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than. |
| VEX.NDS.128.66.0F38 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed qwords in xmm2 and xmm3/m128 for greater than. |

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 second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers.

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.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

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 (VEX.128 encoded version)

DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1, SRC2)
 DEST[255:128] ← 0

PCMPGTB (128-bit Legacy SSE version)

DEST[127:0] ← COMPARE_BYTES_GREATER(DEST, SRC)
 DEST[255:128] (Unmodified)

VPCMPGTW (VEX.128 encoded version)

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1, SRC2)
 DEST[255:128] ← 0

PCMPGTW (128-bit Legacy SSE version)

DEST[127:0] ← COMPARE_WORDS_GREATER(DEST, SRC)
 DEST[255:128] (Unmodified)

VPCMPGTD (VEX.128 encoded version)

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1, SRC2)
 DEST[255:128] ← 0

PCMPGTD (128-bit Legacy SSE version)

DEST[127:0] ← COMPARE_DWORDS_GREATER(DEST, SRC)
 DEST[255:128] (Unmodified)

VPCMPGTQ (VEX.128 encoded version)

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1, SRC2)
 DEST[255:128] ← 0

PCMPGTQ (128-bit Legacy SSE version)

DEST[127:0] ← COMPARE_QWORDS_GREATER(DEST, SRC)
 DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PCMPGTB __m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)

INSTRUCTION SET REFERENCE

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);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

VPERMILPD- Permute Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38 0D /r VPERMILPD xmm1, xmm2, xmm3/m128 | V/V | AVX | Permute double-precision floating-point values in xmm2 using controls from xmm3/mem and store result in xmm1 |
| VEX.NDS.256.66.0F38 0D /r VPERMILPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Permute double-precision floating-point values in ymm2 using controls from ymm3/mem and store result in ymm1 |
| VEX.128.66.0F3A 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8 | V/V | AVX | Permute double-precision floating-point values in xmm2/mem using controls from imm8 |
| VEX.256.66.0F3A 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8 | V/V | AVX | Permute double-precision floating-point values in ymm2/mem using controls from imm8 |

Description

Permute double-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of the second source operand (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

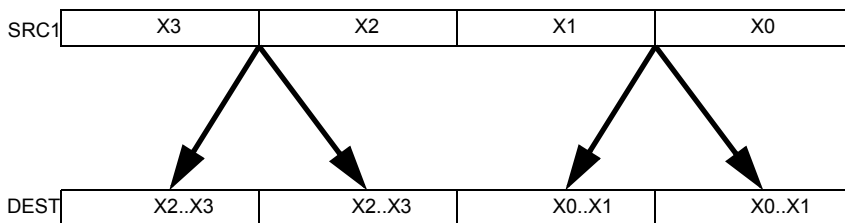


Figure 5-18. VPERMILPD operation

There is one control byte per destination double-precision element. Each control byte is aligned with the low 8 bits of the corresponding double-precision destination element. Each control byte contains a 1-bit select field (see Figure 5-19) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

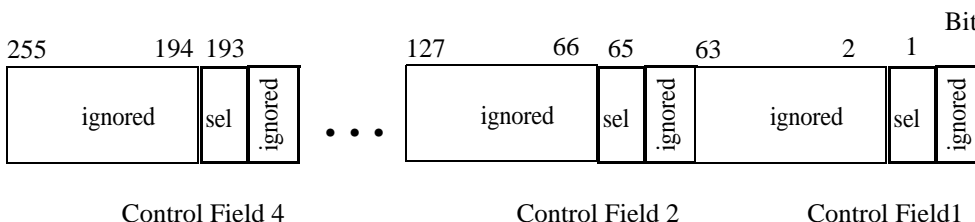


Figure 5-19. VPERMILPD Shuffle Control

(immediate control version)

Permute double-precision floating-point values in the first source operand (second operand) using two, 1-bit control fields in the low 2 bits of the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register.

Note: For the VEX.128.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 05 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (256-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
 IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
 IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
 IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
 IF (imm8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
 IF (imm8[2] = 1) THEN DEST[191:128] ← SRC1[255:192]
 IF (imm8[3] = 0) THEN DEST[255:192] ← SRC1[191:128]
 IF (imm8[3] = 1) THEN DEST[255:192] ← SRC1[255:192]

VPERMILPD (128-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
 IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
 IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
 IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
 DEST[255:128] ← 0

VPERMILPD (256-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
 IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
 IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
 IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
 IF (SRC2[129] = 0) THEN DEST[191:128] ← SRC1[191:128]
 IF (SRC2[129] = 1) THEN DEST[191:128] ← SRC1[255:192]
 IF (SRC2[193] = 0) THEN DEST[255:192] ← SRC1[191:128]
 IF (SRC2[193] = 1) THEN DEST[255:192] ← SRC1[255:192]

VPERMILPD (128-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
 IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
 IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
 IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]

DEST[255:128] ← 0

INSTRUCTION SET REFERENCE

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPD __m128d _mm_permute_pd (__m128d a, int control)

VPERMILPD __m256d _mm256_permute_pd (__m256d a, int control)

VPERMILPD __m128d _mm_permutevar_pd (__m128d a, __m128i control);

VPERMILPD __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6

VPERMIL2PD- Permute 2 Source Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 49 /r/is4 VPERMIL2PD xmm1, xmm2, xmm3/m128, xmm4, imz2 | V/V | AVX | Permute double-precision floating-point values in xmm2 and xmm3/mem. Source data selection is specified in xmm4, results are stored in xmm1 with selective zeroing controls. |
| VEX.NDS.128.66.0F3A.W1 49 /r/is4 VPERMIL2PD xmm1, xmm2, xmm3, xmm4/m128, imz2 | V/V | AVX | Permute double-precision floating-point values in xmm2 and xmm3. Source data selection is specified in xmm4/mem, results are stored in xmm1 with selective zeroing controls. |
| VEX.NDS.256.66.0F3A.W0 49 /r/is4 VPERMIL2PD ymm1, ymm2, ymm3/m256, ymm4, imz2 | V/V | AVX | Permute double-precision floating-point values in ymm2 and ymm3/mem. Source data selection is specified in ymm4, results are stored in ymm1 with selective zeroing controls. |
| VEX.NDS.256.66.0F3A.W1 49 /r/is4 VPERMIL2PD ymm1, ymm2, ymm3, ymm4/m256, imz2 | V/V | AVX | Permute double-precision floating-point values in ymm2 and ymm3. Source data selection is specified in ymm4/mem, results are stored in ymm1 with selective zeroing controls. |

Description

For each quadword element within a 128-bit lane of the destination operand (first operand), select one double-precision floating-point value from two pairs of DP FP values in the same 128-bit lane of the first source operand (2nd operand) and second source operand (third operand) using a source select field specified in the third source operand (fourth operand). Each selected DP FP value is either copied or substituted with zero into the designated quadword element of the destination, depending on the respective match bit in the third source operand and the zero_match control field, imz2, in the immediate operand.

VEX.256 encoded version: If the VEX.W bit is 0, the first and third source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory

location. If the VEX.W bit is 1, the first and second source operand is a YMM register, the third source operand is a YMM register or a 256-bit memory location. The third source operand provide four sets of control fields for each quadword element.

VEX.128 encoded version: If the VEX.W bit is 0, the first and third source operand is a YMM register, the second source operand is a YMM register or a 128-bit memory location. If the VEX.W bit is 1, the first and second source operand is a YMM register, the third source operand is a YMM register or a 128-bit memory location. The third source operand provide four sets of control fields for each quadword element.

The destination operand is always a YMM register. Each set of control field in the third source operand includes:

- a two bit source_select field for each data element i , located at bit offset $(64*i + 1)$, $i = 0, 1$ for VEX.128 encoded version and $i = 0, 1, 2, 3$ for VEX.256 encoded version.
- a match bit located at bit offset $(64*i + 3)$, $i = 0, 1$ for VEX.128 encoded version and $i = 0, 1, 2, 3$ for VEX.256 encoded version.



Figure 5-20. VPERMIL2PD Control Field

The zero_match control field, imz2, is part of the immediate byte (is4) operand. The immediate byte contains:

- a two bit zero_match field (bit 1:0) that controls whether the respective data element of the destination is zeroed or written with the selected DP FP value.
- an encoding field for a source register. In 64-bit mode, bits[7:4] are used to encode the third source register if VEX.W = 0; if VEX.W=1, bits[7:4] encodes the second source register due to source swizzling. In 32-bit mode, bits [6:4] are used to encode the corresponding source register; in 32-bit mode imm8[7] is ignored.

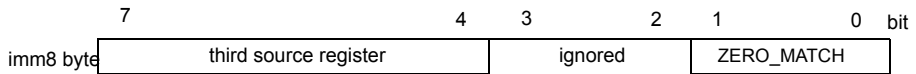


Figure 5-21. VPERMIL2PD Immediate Byte

The zero_match field of the immediate byte and the match bit of the third source operand determine the value that will be written to the corresponding data element.

Table 5-19. VPERMIL2PD Action and ZERO_MATCH Encoding

| zero_match value | VPERMIL2PD Action |
|------------------|--|
| 00b | Selected DP FP value copied to destination element i (Not zero'ed) |
| 01b | Selected DP FP value copied to destination element i (Not zero'ed) |
| 10b | Destination element zeroed if src3 bit[$i*64+3$] is 1 |
| 11b | Destination element zeroed if src3 bit[$i*64+3$] is 0 |

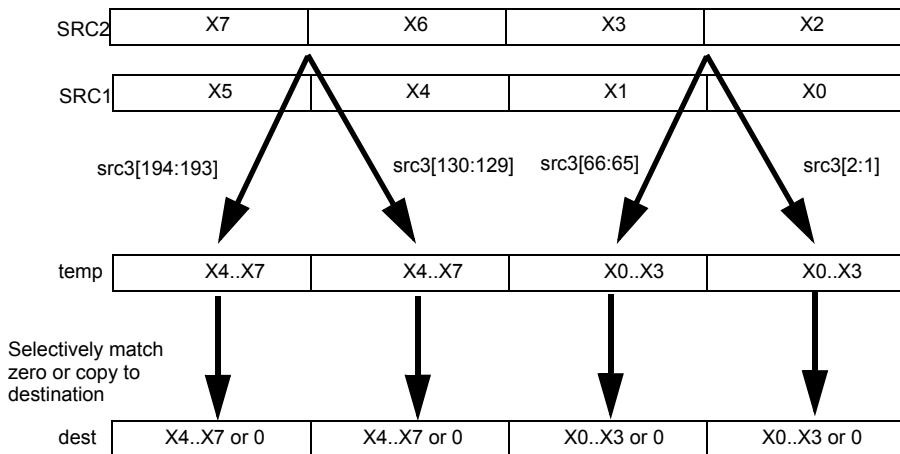


Figure 5-22. VPERMIL2PD operation

Only the 2-bit imz2 field encodes destination match-zero control, compilers and assemblers may implement the following four-operand pseudo-op syntax or four-

operand syntax with constant parameter bounded within the value range between 3 and 0. Compiler can ignore undefined value encoding for `imz2` but should encode the third source register into the immediate byte of the instruction encoding sequence.

Table 5-20 lists the immediate byte encoding that a compiler/assembler can emit for the respective pseudo-op and assembly syntax when `VEX.W = 0`. If `VEX.W = 1`, the second source register, `reg3`, should be encoded in the immediate byte.

Table 5-20. Pseudo-Op and VPERMIL2PD Assembly Support for VEX.W = 0

| Pseudo-Op | VPERMIL2PD Assembly | Is4 byte Encoding |
|--|---|--------------------|
| VPERMILTD2PD <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PD <i>reg1, reg2, reg3, reg4, 0</i> | <i>reg4 00 00B</i> |
| VPERMILTD2PD <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PD <i>reg1, reg2, reg3, reg4, 1</i> | <i>reg4 00 01B</i> |
| VPERMILMO2PD <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PD <i>reg1, reg2, reg3, reg4, 2</i> | <i>reg4 00 10B</i> |
| VPERMILMZ2PD <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PD <i>reg1, reg2, reg3, reg4, 3</i> | <i>reg4 00 11B</i> |

Operation

`select2dp(src1, src2, sel)` // This macro is used by another macro “`sel_and_condzerodp`”

```
{
if (sel[2:1]=0) then TMP ← src1[63:0]
if (sel[2:1]=1) then TMP ← src1[127:64]
if (sel[2:1]=2) then TMP ← src2[63:0]
if (sel[2:1]=3) then TMP ← src2[127:64]
return TMP
}
```

`sel_and_condzerodp(src1, src2, sel)` // This macro is used by `VPERMIL2PD`

```
{
TMP[63:0] ← select2dp(src1[127:0], src2[127:0], sel[2:1])
IF (imm8[1:0] = 2) AND (sel[3]=1) THEN TMP[63:0] ← 0
IF (imm8[1:0] = 3) AND (sel[3]=0) THEN TMP[63:0] ← 0
return TMP
}
```

VPERMIL2PD (VEX.256 encoded version)

```
DEST[63:0] ← sel_and_condzerodp(SRC1[127:0], SRC2[127:0], SRC3[3:1])
DEST[127:64] ← sel_and_condzerodp(SRC1[127:0], SRC2[127:0], SRC3[67:65])
DEST[191:128] ← sel_and_condzerodp(SRC1[255:128], SRC2[255:128], SRC3[131:129])
DEST[255:192] ← sel_and_condzerodp(SRC1[255:128], SRC2[255:128], SRC3[195:193])
```

VPERMIL2PD (128-bit version)

```
DEST[63:0] ← sel_and_condzerodp(SRC1[127:0], SRC2[127:0], SRC3[3:1])
DEST[127:64] ← sel_and_condzerodp(SRC1[127:0], SRC2[127:0], SRC3[67:65])
```

DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMIL2PD __m128d _mm_permute2_pd (__m128d a, __m128d b, __m128i ctrl, int imm)

VPERMIL2PD __m256d _mm256_permute2_pd (__m256d a, __m256d b, __m256i ctrl, int imm)

VSIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6

VPERMILPS- Permute Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F38 0C /r VPERMILPS xmm1, xmm2, xmm3/m128 | V/V | AVX | Permute single-precision floating-point values in xmm2 using controls from xmm3/mem and store result in xmm1 |
| VEX.128.66.0F3A 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8 | V/V | AVX | Permute single-precision floating-point values in xmm2/mem using controls from imm8 and store result in xmm1 |
| VEX.NDS.256.66.0F38 0C /r VPERMILPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Permute single-precision floating-point values in ymm2 using controls from ymm3/mem and store result in ymm1 |
| VEX.256.66.0F3A 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8 | V/V | AVX | Permute single-precision floating-point values in ymm2/mem using controls from imm8 and store result in ymm1 |

Description

(variable control version)

Permute single-precision floating-point values in the first source operand (second operand) using 8-bit control fields in the low bytes of corresponding elements the shuffle control (third operand) and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

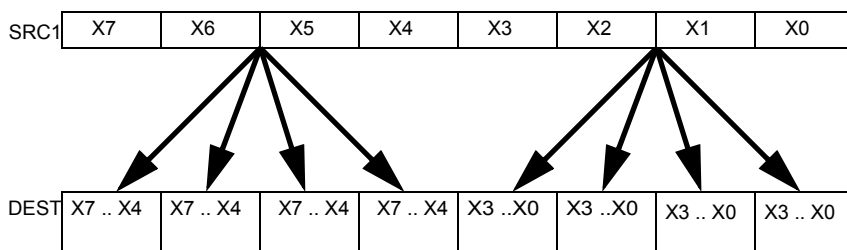


Figure 5-23. VPERMILPS Operation

There is one control byte per destination single-precision element. Each control byte is aligned with the low 8 bits of the corresponding single-precision destination element. Each control byte contains a 2-bit select field (see Figure 5-24) that determines which of the source elements are selected. Source elements are restricted to lie in the same source 128-bit region as the destination.

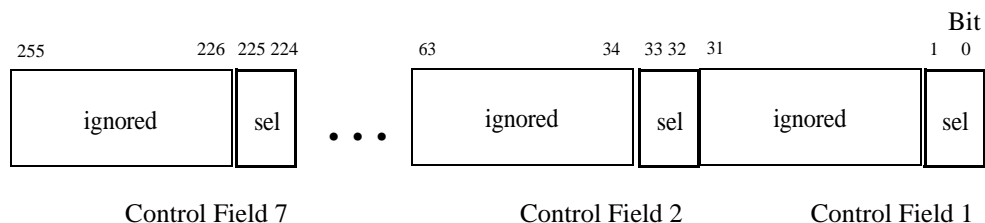


Figure 5-24. VPERMILPS Shuffle Control

(immediate control version)

Permute single-precision floating-point values in the first source operand (second operand) using four 2-bit control fields in the 8-bit immediate and store results in the destination operand (first operand). The source operand is a YMM register or 256-bit memory location and the destination operand is a YMM register. This is similar to a wider version of PSHUFD, just operating on single-precision floating-point values.

Note: For the VEX.128.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the VEX.256.66.0F3A 04 instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP ← SRC[31:0];
  1:  TMP ← SRC[63:32];
  2:  TMP ← SRC[95:64];
  3:  TMP ← SRC[127:96];
ESAC;
RETURN TMP
}
```

VPERMILPS (256-bit immediate version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[255:128] ← 0
```

VPERMILPS (256-bit variable version)

```
DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ← Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ← Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ← Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);
```

VPERMILPS (128-bit variable version)

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
 DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
 DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
 DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
 DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS __m128 _mm_permute_ps (__m128 a, int control);
 VPERMILPS __m256 _mm256_permute_ps (__m256 a, int control);
 VPERMILPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);
 VPERMILPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6

VPERMIL2PS- Permute 2 Source Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 48 /r/is4 VPERMIL2PS xmm1, xmm2, xmm3/m128, xmm4, imz2 | V/V | AVX | Permute single-precision floating-point values in xmm2 and xmm3/mem. Source data selection is specified in xmm4, results are stored in xmm1 with selective zeroing controls. |
| VEX.NDS.128.66.0F3A.W1 48 /r/is4 VPERMIL2PS xmm1, xmm2, xmm3, xmm4/m128, imz2 | V/V | AVX | Permute single-precision floating-point values in xmm2 and xmm3. Source data selection is specified in xmm4/mem, results are stored in xmm1 with selective zeroing controls. |
| VEX.NDS.256.66.0F3A.W0 48 /r/is4 VPERMIL2PS ymm1, ymm2, ymm3/m256, ymm4, imz2 | V/V | AVX | Permute single-precision floating-point values in ymm2 and ymm3/mem. Source data selection is specified in ymm4, results are stored in ymm1 with selective zeroing controls. |
| VEX.NDS.256.66.0F3A.W1 48 /r/is4 VPERMIL2PS ymm1, ymm2, ymm3, ymm4/m256, imz2 | V/V | AVX | Permute single-precision floating-point values in ymm2 and ymm3. Source data selection is specified in ymm4/mem, results are stored in ymm1 with selective zeroing controls. |

Description

For each dword element within a 128-bit lane of the destination operand (first operand), select one single-precision floating-point value from four pairs of SP FP values in the same 128-bit lane of the first source operand (2nd operand) and second source operand (third operand) using a source select field specified in the third source operand (fourth operand). Each selected SP FP value is either copied or substituted with zero into the designated dword element of the destination, depending on the respective match bit in the third source operand and the zero_match control field, imz2, in the immediate operand.

VEX.256 encoded version: If the VEX.W bit is 0, the first and third source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location. If the VEX.W bit is 1, the first and second source operand is a YMM register, the third source operand is a YMM register or a 256-bit memory location. The third source operand provide four sets of control fields for each quadword element.

VEX.128 encoded version: If the VEX.W bit is 0, the first and third source operand is a YMM register, the second source operand is a YMM register or a 128-bit memory location. If the VEX.W bit is 1, the first and second source operand is a YMM register, the third source operand is a YMM register or a 128-bit memory location. The third source operand provide four sets of control fields for each quadword element.

The destination operand is always a YMM register. Each set of control field in the third source operand includes:

- a two bit source_select field for each data element i , located at bit offset $(32*i + 1)$, $i = 0, 1, 2, 3$ for VEX.128 encoded version and $i = 0, 1, 2, 3, 4, 5, 6, 7$ for VEX.256 encoded version.
- a match bit located at bit offset $(32*i + 3)$, $i = 0, 1, 2, 3$ for VEX.128 encoded version and $i = 0, 1, 2, 3, 4, 5, 6, 7$ for VEX.256 encoded version.

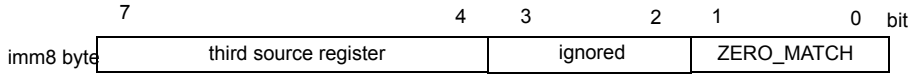


Figure 5-25. VPERMIL2PS Control Field

The zero_match control field, imz2, is part of the immediate byte operand. The immediate byte (is4) contains:

- a two bit zero_match field (bit 1:0) that controls whether the respective data element of the destination is zeroed or written with the selected DP FP value.
- an encoding field for a source register. In 64-bit mode, bits[7:4] are used to encode the third source register if VEX.W = 0; if VEX.W=1, bits[7:4] encodes the second source register due to source swizzling. In 32-bit mode, bits [6:4] are used to encode the corresponding source register; in 32-bit mode imm8[7] is ignored.

Figure 5-26. VPERMIL2PS Immediate Byte



The zero_match field of the immediate byte and the match bit of the third source operand determine the value that will be written to the corresponding data element.

Table 5-21. VPERMIL2PS Action and ZERO_MATCH Encoding

| zero_match value | VPERMIL2PS Action |
|------------------|---|
| 00b | Selected SP FP value copied to destination element i (Not zeroed) |
| 01b | Selected SP FP value copied to destination element i (Not zeroed) |
| 10b | Destination element zeroed if src3 bit[i*64+3] is 1 |
| 11b | Destination element zeroed if src3 bit[i*64+3] is 0 |

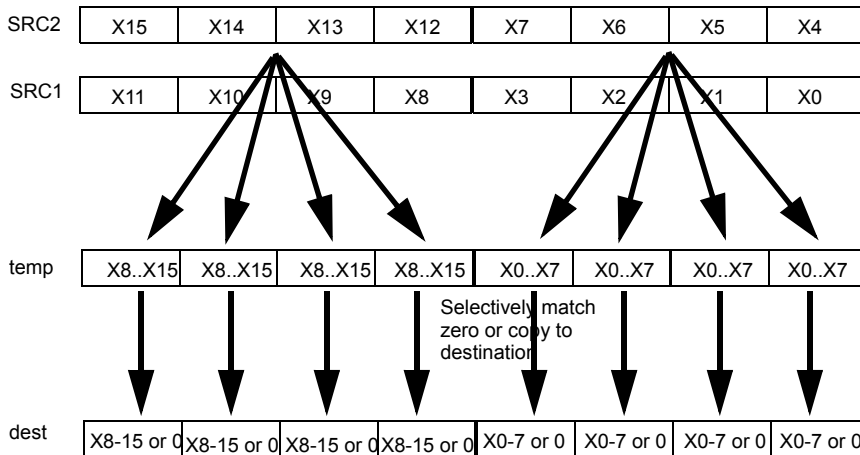


Figure 5-27. VPERMIL2PS operation

Only the 2-bit `imz2` field encodes destination match-zero control, compilers and assemblers may implement the following four-operand pseudo-ops syntax or four-operand syntax with constant parameter bounded within the value range between 3 and 0. Compiler can ignore undefined value encoding for `imz2` but should encode the third source register into the immediate byte of the instruction encoding sequence.

Table 5-22 lists the immediate byte encoding that a compiler/assembler can emit for the respective pseudo-op and assembly syntax when `VEX.W = 0`. If `VEX.W = 1`, the second source register, `reg3`, should be encoded in the immediate byte.

Table 5-22. Pseudo-Op and VPERMIL2PS Assembly Support for VEX.W = 0

| Pseudo-Op | VPERMIL2PS Assembly | Imm8 Encoding |
|--|---|--------------------|
| VPERMILTD2PS <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PS <i>reg1, reg2, reg3, reg4, 0</i> | <i>reg4 00 00B</i> |
| VPERMILTD2PS <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PS <i>reg1, reg2, reg3, reg4, 1</i> | <i>reg4 00 01B</i> |
| VPERMILMO2PS <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PS <i>reg1, reg2, reg3, reg4, 2</i> | <i>reg4 00 10B</i> |
| VPERMILMZ2PS <i>reg1, reg2, reg3, reg4</i> | VPERMIL2PS <i>reg1, reg2, reg3, reg4, 3</i> | <i>reg4 00 11B</i> |

Operation

`select2sp(src1, src2, sel)` // This macro is used by another macro “`sel_and_condzersp`” below

```
{
if (sel[2:0]=0) then TMP ← src1[31:0]
if (sel[2:0]=1) then TMP ← src1[63:32]
if (sel[2:0]=2) then TMP ← src1[95:64]
if (sel[2:0]=3) then TMP ← src1[127:96]
if (sel[2:0]=4) then TMP ← src2[31:0]
if (sel[2:0]=5) then TMP ← src2[63:32]
if (sel[2:0]=6) then TMP ← src2[95:64]
if (sel[2:0]=7) then TMP ← src2[127:96]
return TMP
}
```

`sel_and_condzersp(src1, src2, sel)` // This macro is used by VPERMIL2PS

```
{
TMP[31:0] ← select2sp(src1[127:0], src2[127:0], sel[2:0])
IF (imm8[1:0] = 2) AND (sel[3]=1) THEN TMP[31:0] ← 0
IF (imm8[1:0] = 3) AND (sel[3]=0) THEN TMP[31:0] ← 0
return TMP
}
```

VPERMIL2PS (VEX.256 encoded version)

`DEST[31:0] ← sel_and_condzersp(SRC1[127:0], SRC2[127:0], SRC3[3:0])`

INSTRUCTION SET REFERENCE

DEST[63:32] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[35:32])
DEST[95:64] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[67:64])
DEST[127:96] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[99:96])
DEST[159:128] ← sel_and_condzosp(SRC1[255:128], SRC2[255:128], SRC3[131:128])
DEST[191:160] ← sel_and_condzosp(SRC1[255:128], SRC2[255:128], SRC3[163:160])
DEST[223:192] ← sel_and_condzosp(SRC1[255:128], SRC2[255:128], SRC3[195:192])
DEST[255:224] ← sel_and_condzosp(SRC1[255:128], SRC2[255:128], SRC3[227:224])

VPERMIL2PS (128-bit version)

DEST[31:0] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[3:1])
DEST[63:32] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[35:32])
DEST[95:64] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[67:64])
DEST[127:96] ← sel_and_condzosp(SRC1[127:0], SRC2[127:0], SRC3[99:96])
DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMIL2PS __m128 __mm_permute2_ps (__m128 a, __m128 b, __m128i ctrl, int imm)

VPERMIL2PS __m256 __mm256_permute2_ps (__m256 a, __m256 b, __m256i ctrl, int imm)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6

VPERM2F128- Permute Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.256.66.0F3A 06 /r ib VPERM2F128 ymm1, ymm2, ymm3/m256, imm8 | V/V | AVX | Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1 |

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

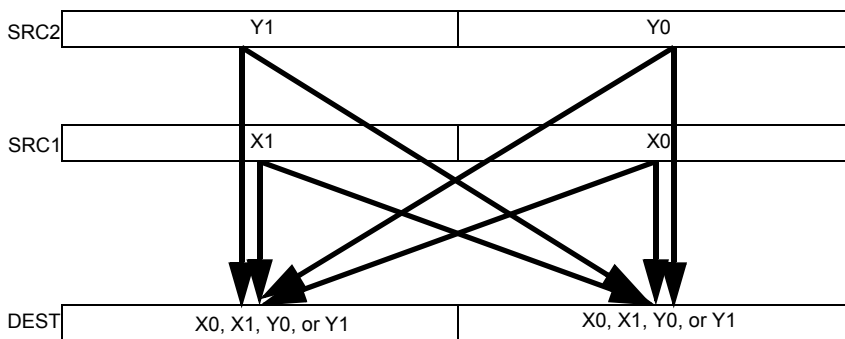


Figure 5-28. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2F128

CASE IMM8[1:0] of

0: DEST[127:0] ← SRC1[127:0]

1: DEST[127:0] ← SRC1[255:128]

2: DEST[127:0] ← SRC2[127:0]

3: DEST[127:0] ← SRC2[255:128]

ESAC

CASE IMM8[5:4] of

0: DEST[255:128] ← SRC1[127:0]

1: DEST[255:128] ← SRC1[255:128]

2: DEST[255:128] ← SRC2[127:0]

3: DEST[255:128] ← SRC2[255:128]

ESAC

IF (imm8[3])

DEST[127:0] ← 0

FI

IF (imm8[7])

DEST[255:128] ← 0

FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128 __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)

VPERM2F128 __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)

VPERM2F128 __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD If VEX.L = 0.

PEXTRB/PEXTRW/PEXTRD/PEXTRQ- Extract Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 3A 14 /r ib PEXTRB reg/m8, xmm2, imm8 | 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 PEXTRW reg, xmm1, imm8 | 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 PEXTRW reg/m16, xmm2, imm8 | 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 PEXTRD r32/m32, xmm2, imm8 | 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 PEXTRQ r64/m64, xmm2, imm8 | N.E./V | 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 VPEXTRB reg/m8, xmm2, imm8 | 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 VPEXTRW reg, xmm1, imm8 | 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. |

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| VEX.128.66.0F3A.15 /r ib VPEXTRW reg/m16, xmm2, imm8 | 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 VPEXTRD r32/m32, xmm2, imm8 | V/V | AVX | Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32. |
| VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8 | N.E./V | AVX | Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64. |

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 \leftarrow Imm8[2:0]DEST[15:0] \leftarrow ((Src >> Src_Offset*16) AND 0FFFFh)DEST[63:16] \leftarrow ZERO_FILL;

ELSE

SRC_Offset \leftarrow Imm8[2:0]DEST[15:0] \leftarrow ((Src >> Src_Offset*16) AND 0FFFFh)DEST[31:16] \leftarrow ZERO_FILL;

FI

(V)PEXTRB (dest=m8)SRC_Offset \leftarrow Imm8[3:0]Mem8 \leftarrow (Src >> Src_Offset*8)**(V)PEXTRB (dest=reg)**

IF (64-Bit Mode)

THEN

SRC_Offset \leftarrow Imm8[3:0]DEST[7:0] \leftarrow ((Src >> Src_Offset*8) AND 0FFh)DEST[63:8] \leftarrow ZERO_FILL;

ELSE

SRC_Offset \leftarrow Imm8[3:0];DEST[7:0] \leftarrow ((Src >> 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

See Exceptions Type 5; additionally

INSTRUCTION SET REFERENCE

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

PHADDW/PHADD - Packed Horizontal Add

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 38 01 /r PHADDW xmm1, xmm2/m128 | V/V | SSSE3 | Add 16-bit signed integers horizontally, pack to xmm1. |
| 66 0F 38 02 /r PHADD xmm1, xmm2/m128 | V/V | SSSE3 | Add 32-bit signed integers horizontally, pack to xmm1. |
| VEX.NDS.128.66.0F38 01 /r VPHADDW xmm1, xmm2, xmm3/m128 | V/V | AVX | Add 16-bit signed integers horizontally, pack to xmm1. |
| VEX.NDS.128.66.0F38 02 /r VPHADD xmm1, xmm2, xmm3/m128 | V/V | AVX | Add 32-bit signed integers horizontally, pack to xmm1. |

Description

PHADDW adds two adjacent 16-bit signed integers horizontally from the second source operand and the first source operand and packs the 16-bit signed results to the destination operand. PHADD adds two adjacent 32-bit signed integers horizontally from the second source operand and the first source operand and packs the 32-bit signed results to the destination operand. The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHADDW (VEX.128 encoded version)

```

DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]

```

INSTRUCTION SET REFERENCE

DEST[255:128] ← 0

VPHADD (VEX.128 encoded version)

DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[255:128] ← 0

PHADDW (128-bit Legacy SSE version)

DEST[15:0] ← DEST[31:16] + DEST[15:0]
DEST[31:16] ← DEST[63:48] + DEST[47:32]
DEST[47:32] ← DEST[95:80] + DEST[79:64]
DEST[63:48] ← DEST[127:112] + DEST[111:96]
DEST[79:64] ← SRC[31:16] + SRC[15:0]
DEST[95:80] ← SRC[63:48] + SRC[47:32]
DEST[111:96] ← SRC[95:80] + SRC[79:64]
DEST[127:112] ← SRC[127:112] + SRC[111:96]
DEST[255:128] (Unmodified)

PHADD (128-bit Legacy SSE version)

DEST[31:0] ← DEST[63:32] + DEST[31:0]
DEST[63:32] ← DEST[127:96] + DEST[95:64]
DEST[95:64] ← SRC[63:32] + SRC[31:0]
DEST[127:96] ← SRC[127:96] + SRC[95:64]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHADDW `__m128i _mm_hadd_epi16 (__m128i a, __m128i b)`

PHADD `__m128i _mm_hadd_epi32 (__m128i a, __m128i b)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHADDSW - Packed Horizontal Add with Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 38 03 /r PHADDSW xmm1, xmm2/m128 | V/V | SSSE3 | Add 16-bit signed integers horizontally, pack saturated integers to xmm1. |
| VEX.NDS.128.66.0F38 03 /r VPHADDSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Add 16-bit signed integers horizontally, pack saturated integers to xmm1. |

Description

PHADDSW adds two adjacent signed 16-bit integers horizontally from the second source and first source operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand. The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHADDSW (VEX.128 encoded version)

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[255:128] ← 0

```

PHADDSW (128-bit Legacy SSE version)

```

DEST[15:0] = SaturateToSignedWord(DEST[31:16] + DEST[15:0])
DEST[31:16] = SaturateToSignedWord(DEST[63:48] + DEST[47:32])
DEST[47:32] = SaturateToSignedWord(DEST[95:80] + DEST[79:64])
DEST[63:48] = SaturateToSignedWord(DEST[127:112] + DEST[111:96])

```

INSTRUCTION SET REFERENCE

DEST[79:64] = SaturateToSignedWord(SRC[31:16] + SRC[15:0])
DEST[95:80] = SaturateToSignedWord(SRC[63:48] + SRC[47:32])
DEST[111:96] = SaturateToSignedWord(SRC[95:80] + SRC[79:64])
DEST[127:112] = SaturateToSignedWord(SRC[127:112] + SRC[111:96])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW __m128i _mm_hadds_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHMINPOSUW - Horizontal Minimum and Position

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 41 /r PHMINPOSUW xmm1, xmm2/m128 | V/V | SSE4_1 | Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1 |
| VEX.128.66.0F38 41 /r VPHMINPOSUW xmm1, xmm2/m128 | V/V | AVX | Find the minimum unsigned word in xmm2/m128 and place its value in the low word of xmm1 and its index in the second-lowest word of xmm1 |

Description

Determine the minimum unsigned word value in the source operand and place the unsigned word in the low word (bits 0-15) of the destination operand. The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHMINPOSUW (VEX.128 encoded version)

INDEX \leftarrow 0

MIN \leftarrow SRC[15:0]

IF (SRC[31:16] < MIN) THEN INDEX \leftarrow 1; MIN \leftarrow SRC[31:16]

IF (SRC[47:32] < MIN) THEN INDEX \leftarrow 2; MIN \leftarrow SRC[47:32]

* Repeat operation for words 3 through 6

IF (SRC[127:112] < MIN) THEN INDEX \leftarrow 7; MIN \leftarrow SRC[127:112]

DEST[15:0] \leftarrow MIN

DEST[18:16] \leftarrow INDEX

DEST[127:19] \leftarrow 00000000000000000000000000000000H

DEST[255:128] \leftarrow 0

INSTRUCTION SET REFERENCE

PHMINPOSUW (128-bit Legacy SSE version)

INDEX \leftarrow 0

MIN \leftarrow SRC[15:0]

IF (SRC[31:16] < MIN) THEN INDEX \leftarrow 1; MIN \leftarrow SRC[31:16]

IF (SRC[47:32] < MIN) THEN INDEX \leftarrow 2; MIN \leftarrow SRC[47:32]

* Repeat operation for words 3 through 6

IF (SRC[127:112] < MIN) THEN INDEX \leftarrow 7; MIN \leftarrow SRC[127:112]

DEST[15:0] \leftarrow MIN

DEST[18:16] \leftarrow INDEX

DEST[127:19] \leftarrow 00000000000000000000000000000000H

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW __m128i _mm_minpos_epu16(__m128i packed_words)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD

If VEX.L = 1.

If VEX.vvvv \neq 1111B.

PHSUBW/PHSUBD - Packed Horizontal Subtract

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 05 /r PHSUBW xmm1, xmm2/m128 | V/V | SSSE3 | Subtract 16-bit signed integers horizontally, pack to xmm1. |
| 66 0F 38 06 /r PHSUBD xmm1, xmm2/m128 | V/V | SSSE3 | Subtract 32-bit signed integers horizontally, pack to xmm1. |
| VEX.NDS.128.66.0F38 05 /r VPHSUBW xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract 16-bit signed integers horizontally, pack to xmm1. |
| VEX.NDS.128.66.0F38 06 /r VPHSUBD xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract 32-bit signed integers horizontally, pack to xmm1. |

Description

PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the second source operand and destination operands, and packs the signed 16-bit results to the destination operand. PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand.

The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHSUBW (VEX.128 encoded version)

DEST[15:0] ← SRC1[15:0] - SRC1[31:16]

DEST[31:16] ← SRC1[47:32] - SRC1[63:48]

DEST[47:32] ← SRC1[79:64] - SRC1[95:80]

DEST[63:48] ← SRC1[111:96] - SRC1[127:112]

INSTRUCTION SET REFERENCE

DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[255:128] ← 0

VPHSUBD (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[255:128] ← 0

PHSUBW (128-bit Legacy SSE version)

DEST[15:0] ← DEST[15:0] - DEST[31:16]
DEST[31:16] ← DEST[47:32] - DEST[63:48]
DEST[47:32] ← DEST[79:64] - DEST[95:80]
DEST[63:48] ← DEST[111:96] - DEST[127:112]
DEST[79:64] ← SRC[15:0] - SRC[31:16]
DEST[95:80] ← SRC[47:32] - SRC[63:48]
DEST[111:96] ← SRC[79:64] - SRC[95:80]
DEST[127:112] ← SRC[111:96] - SRC[127:112]
DEST[255:128] (Unmodified)

PHSUBD (128-bit Legacy SSE version)

DEST[31:0] ← DEST[31:0] - DEST[63:32]
DEST[63:32] ← DEST[95:64] - DEST[127:96]
DEST[95:64] ← SRC[31:0] - SRC[63:32]
DEST[127:96] ← SRC[95:64] - SRC[127:96]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBW __m128i _mm_hsub_epi16 (__m128i a, __m128i b)

PHSUBD __m128i _mm_hsub_epi32 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PHSUBSW - Packed Horizontal Subtract with Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 38 07 /r PHSUBSW xmm1, xmm2/m128 | V/V | SSSE3 | Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1 |
| VEX.NDS.128.66.0F38 07 /r VPHSUBSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1 |

Description

PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the second source and first source operands. The signed, saturated 16-bit results are packed to the destination operand. The destination and first source operand are XMM registers. The second operand can be an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPHSUBSW (VEX.128 encoded version)

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[255:128] ← 0

```

PHSUBSW (128-bit Legacy SSE version)

```

DEST[15:0] = SaturateToSignedWord(DEST[15:0] - DEST[31:16])
DEST[31:16] = SaturateToSignedWord(DEST[47:32] - DEST[63:48])
DEST[47:32] = SaturateToSignedWord(DEST[79:64] - DEST[95:80])

```

INSTRUCTION SET REFERENCE

DEST[63:48] = SaturateToSignedWord(DEST[111:96] - DEST[127:112])
DEST[79:64] = SaturateToSignedWord(SRC[15:0] - SRC[31:16])
DEST[95:80] = SaturateToSignedWord(SRC[47:32] - SRC[63:48])
DEST[111:96] = SaturateToSignedWord(SRC[79:64] - SRC[95:80])
DEST[127:112] = SaturateToSignedWord(SRC[SRC[111:96] - 127:112])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PINSRB/PINSRW/PINSRD/PINSRQ- Insert Integer

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 20 /r ib PINSRB xmm1, r32/m8, imm8 | 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 PINSRW xmm1, r32/m16, imm8 | 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 PINSRD xmm1, r32/m32, imm8 | 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 PINSRQ xmm1, r64/m64, imm8 | N.E./V | 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 VPINSRB xmm1, xmm2, r32/m8, imm8 | 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 VPINSRW xmm1, xmm2, r32/m16, imm8 | 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 VPINSRD xmm1, xmm2, r32/m32, imm8 | V/V | AVX | Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8 |
| VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8 | N.E./V | AVX | Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in 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.

INSTRUCTION SET REFERENCE

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM 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 (VEX.128 encoded version)

```

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

```

VPINSRD (VEX.128 encoded version)

```

SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

```

VPINSRW (VEX.128 encoded version)

```

SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[255:128] ← 0

```


INSTRUCTION SET REFERENCE

VPINSRB (VEX.128 encoded version)

SEL \leftarrow imm8[3:0]
DEST[127:0] \leftarrow write_b_element(SEL, SRC2, SRC1)
DEST[255:128] \leftarrow 0

PINSRQ (Legacy SSE version)

SEL \leftarrow imm8[0]
DEST[127:0] \leftarrow write_q_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

PINSRD (Legacy SSE version)

SEL \leftarrow imm8[1:0]
DEST[127:0] \leftarrow write_d_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

PINSRW (Legacy SSE version)

SEL \leftarrow imm8[2:0]
DEST[127:0] \leftarrow write_w_element(SEL, SRC, DEST)
DEST[255:128] (Unmodified)

PINSRB (Legacy SSE version)

SEL \leftarrow imm8[3:0]
DEST[127:0] \leftarrow write_b_element(SEL, SRC, DEST)
DEST[255:128] (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

See Exceptions Type 5; additionally

#UD If VEX.L = 1.

PMADDWD- Multiply and Add Packed Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F F5 /r PMADDWD xmm1, xmm2/m128 | 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 VPMADDWD xmm1, xmm2, xmm3/m128 | 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. |

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 second source operand is an XMM register or a 128-bit memory location.

The first source and destination operands are XMM registers. 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: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMADDWD (VEX.128 encoded version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[15:0] * \text{SRC2}[15:0]) + (\text{SRC1}[31:16] * \text{SRC2}[31:16])$$

$$\text{DEST}[63:32] \leftarrow (\text{SRC1}[47:32] * \text{SRC2}[47:32]) + (\text{SRC1}[63:48] * \text{SRC2}[63:48])$$

$$\text{DEST}[95:64] \leftarrow (\text{SRC1}[79:64] * \text{SRC2}[79:64]) + (\text{SRC1}[95:80] * \text{SRC2}[95:80])$$

$$\text{DEST}[127:96] \leftarrow (\text{SRC1}[111:96] * \text{SRC2}[111:96]) + (\text{SRC1}[127:112] * \text{SRC2}[127:112])$$

$$\text{DEST}[255:128] \leftarrow 0$$

INSTRUCTION SET REFERENCE

PMADDWD (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16])$

$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48])$

$\text{DEST}[95:64] \leftarrow (\text{DEST}[79:64] * \text{SRC}[79:64]) + (\text{DEST}[95:80] * \text{SRC}[95:80])$

$\text{DEST}[127:96] \leftarrow (\text{DEST}[111:96] * \text{SRC}[111:96]) + (\text{DEST}[127:112] * \text{SRC}[127:112])$

$\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

`PMADDWD __m128i _mm_madd_epi16 (__m128i a, __m128i b)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMADDUBSW- Multiply and Add Packed Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128 | 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 VPMADDUBSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1. |

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. The second source operand can be an XMM register or 128-bit memory location. The first source operand and destination operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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[255:128] ← 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

INSTRUCTION SET REFERENCE

DEST[127:112] ← SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]*
DEST[119:112]);
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMADDUBSW __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXSB/PMAXSW/PMAXSD- Maximum of Packed Signed Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 3C /r PMAXSB xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1. |
| 66 0F EE /r PMAXSW xmm1, xmm2/m128 | V/V | SSE2 | Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1. |
| 66 0F 38 3D /r PMAXSD xmm1, xmm2/m128 | 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 3C /r VPMAXSB xmm1, xmm2, xmm3/m128 | 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 EE /r VPMAXSW xmm1, xmm2, xmm3/m128 | 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 3D /r VPMAXSD xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1. |

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 maximum value for each pair of integers to the destination operand. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

INSTRUCTION SET REFERENCE

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXSB (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[255:128] (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[255:128] ← 0

PMAXSW (128-bit Legacy SSE version)

IF DEST[15:0] > SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF DEST[127:112] > SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[255:128] (Unmodified)

VPMAXSW (VEX.128 encoded version)

IF SRC1[15:0] > SRC2[15:0] THEN

DEST[15:0] ← SRC1[15:0];

ELSE

DEST[15:0] \leftarrow 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] \leftarrow SRC1[127:112];

ELSE

DEST[127:112] \leftarrow SRC2[127:112]; FI;

DEST[255:128] \leftarrow 0

PMAXSD (128-bit Legacy SSE version)

IF DEST[31:0] > SRC[31:0] THEN

DEST[31:0] \leftarrow DEST[31:0];

ELSE

DEST[31:0] \leftarrow SRC[31:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF DEST[127:95] > SRC[127:95] THEN

DEST[127:95] \leftarrow DEST[127:95];

ELSE

DEST[127:95] \leftarrow SRC[127:95]; FI;

DEST[255:128] (Unmodified)

VPMAXSD (VEX.128 encoded version)

IF SRC1[31:0] > SRC2[31:0] THEN

DEST[31:0] \leftarrow SRC1[31:0];

ELSE

DEST[31:0] \leftarrow SRC2[31:0]; FI;

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

IF SRC1[127:95] > SRC2[127:95] THEN

DEST[127:95] \leftarrow SRC1[127:95];

ELSE

DEST[127:95] \leftarrow SRC2[127:95]; FI;

DEST[255:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);

PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)

PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMAXUB/PMAXUW/PMAXUD- Maximum of Packed Unsigned Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F DE /r PMAXUB xmm1, xmm2/m128 | V/V | SSE2 | Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1. |
| 66 0F 38 3E/r PMAXUW xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1. |
| 66 0F 38 3F /r PMAXUD xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1. |
| VEX.NDS.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128 | 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 VPMAXUW xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1. |
| VEX.NDS.128.66.0F38 3F /r VPMAXUD xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1. |

Description

Performs a SIMD compare of the packed unsigned byte, word, or dword 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. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

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[255:128] (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[255:128] ← 0

PMAXUW (128-bit Legacy SSE version)

IF DEST[15:0] > SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF DEST[127:112] > SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[255:128] (Unmodified)

VPMAXUW (VEX.128 encoded version)

IF SRC1[15:0] > SRC2[15:0] THEN

DEST[15:0] ← SRC1[15:0];

ELSE

DEST[15:0] \leftarrow 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] \leftarrow SRC1[127:112];

ELSE

DEST[127:112] \leftarrow SRC2[127:112]; FI;

DEST[255:128] \leftarrow 0

PMAXUD (128-bit Legacy SSE version)

IF DEST[31:0] > SRC[31:0] THEN

DEST[31:0] \leftarrow DEST[31:0];

ELSE

DEST[31:0] \leftarrow SRC[31:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF DEST[127:95] > SRC[127:95] THEN

DEST[127:95] \leftarrow DEST[127:95];

ELSE

DEST[127:95] \leftarrow SRC[127:95]; FI;

DEST[255:128] (Unmodified)

VPMAXUD (VEX.128 encoded version)

IF SRC1[31:0] > SRC2[31:0] THEN

DEST[31:0] \leftarrow SRC1[31:0];

ELSE

DEST[31:0] \leftarrow SRC2[31:0]; FI;

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

IF SRC1[127:95] > SRC2[127:95] THEN

DEST[127:95] \leftarrow SRC1[127:95];

ELSE

DEST[127:95] \leftarrow SRC2[127:95]; FI;

DEST[255:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB __m128i _mm_max_epu8 (__m128i a, __m128i b);

PMAXUW __m128i _mm_max_epu16 (__m128i a, __m128i b)

PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINSB/PMINSW/PMINSD- Minimum of Packed Signed Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 38 /r PMINSB xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1. |
| 66 0F EA /r PMINSW xmm1, xmm2/m128 | V/V | SSE2 | Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1. |
| 66 0F 38 39 /r PMINSD xmm1, xmm2/m128 | 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 38 /r VPMINSB xmm1, xmm2, xmm3/m128 | 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 VPMINSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1. |
| VEX.NDS.128.66.0F38 39 /r VPMINSD xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1. |

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. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

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[255:128] (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[255:128] ← 0

PMINSW (128-bit Legacy SSE version)

IF DEST[15:0] < SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF DEST[127:112] < SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[255:128] (Unmodified)

VPMINSW (VEX.128 encoded version)

IF SRC1[15:0] < SRC2[15:0] THEN

DEST[15:0] ← SRC1[15:0];

ELSE

DEST[15:0] \leftarrow 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] \leftarrow SRC1[127:112];

ELSE

DEST[127:112] \leftarrow SRC2[127:112]; FI;

DEST[255:128] \leftarrow 0

PMINSD (128-bit Legacy SSE version)

IF DEST[31:0] < SRC[31:0] THEN

DEST[31:0] \leftarrow DEST[31:0];

ELSE

DEST[31:0] \leftarrow SRC[31:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

IF DEST[127:95] < SRC[127:95] THEN

DEST[127:95] \leftarrow DEST[127:95];

ELSE

DEST[127:95] \leftarrow SRC[127:95]; FI;

DEST[255:128] (Unmodified)

VPMINSD (VEX.128 encoded version)

IF SRC1[31:0] < SRC2[31:0] THEN

DEST[31:0] \leftarrow SRC1[31:0];

ELSE

DEST[31:0] \leftarrow SRC2[31:0]; FI;

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

IF SRC1[127:95] < SRC2[127:95] THEN

DEST[127:95] \leftarrow SRC1[127:95];

ELSE

DEST[127:95] \leftarrow SRC2[127:95]; FI;

DEST[255:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);

PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)

PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

INSTRUCTION SET REFERENCE

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMINUB/PMINUW/PMINUD- Minimum of Packed Unsigned Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F DA /r PMINUB xmm1, xmm2/m128 | V/V | SSE2 | Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1. |
| 66 0F 38 3A/r PMINUW xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1. |
| 66 0F 38 3B /r PMINUD xmm1, xmm2/m128 | V/V | SSE4_1 | Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1. |
| VEX.NDS.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128 | 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 VPMINUW xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1. |
| VEX.NDS.128.66.0F38 3B /r VPMINUD xmm1, xmm2, xmm3/m128 | V/V | AVX | Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1. |

Description

Performs a SIMD compare of the packed unsigned 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. The first source and destination operand is an XMM register; The second source operand is an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

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[255:128] (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[255:128] ← 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[255:128] (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[255:128] ← 0

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[127:95] < SRC[127:95] THEN

DEST[127:95] ← DEST[127:95];

ELSE

DEST[127:95] ← SRC[127:95]; FI;

DEST[255:128] (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[127:95] < SRC2[127:95] THEN

DEST[127:95] ← SRC1[127:95];

ELSE

DEST[127:95] ← SRC2[127:95]; FI;

DEST[255:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)

INSTRUCTION SET REFERENCE

PMINUW `__m128i _mm_min_epu16 (__m128i a, __m128i b);`

PMINUD `__m128i _mm_min_epu32 (__m128i a, __m128i b);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMOVMSKB- Move Byte Mask

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F D7 /r PMOVMSKB reg, xmm1 | V/V | SSE2 | Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros. |
| VEX.128.66.0F D7 /r VPMOVMSKB reg, xmm1 | V/V | AVX | Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros. |

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand). The source operand is an XMM register; the destination operand is a general-purpose register. The byte mask is 16-bits.

The destination operand is a general-purpose register. In 64-bit mode, the default operand size of the destination operand is 64 bits. The upper bits above bit 15 are filled with zeros. REX.W is ignored.

VEX.128 encodings are valid but identical in function. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

(V)PMOVMSKB instruction with 128-bit source operand and r32:

$r32[0] \leftarrow \text{SRC}[7];$

$r32[1] \leftarrow \text{SRC}[15];$

(* Repeat operation for bytes 2 through 14 *)

$r32[15] \leftarrow \text{SRC}[127];$

$r32[31:16] \leftarrow \text{ZERO_FILL};$

(V)PMOVMSKB instruction with 128-bit source operand and r64:

$r64[0] \leftarrow \text{SRC}[7];$

$r64[1] \leftarrow \text{SRC}[15];$

(* Repeat operation for bytes 2 through 14 *)

$r64[15] \leftarrow \text{SRC}[127];$

$r64[63:16] \leftarrow \text{ZERO_FILL};$

INSTRUCTION SET REFERENCE

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB int _mm_movemask_epi8 (__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

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

PMOVSX - Packed Move with Sign Extend

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64 | V/V | SSE4_1 | Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1 |
| 66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32 | V/V | SSE4_1 | Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 |
| 66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16 | V/V | SSE4_1 | Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 |
| 66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64 | V/V | SSE4_1 | Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 |
| 66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32 | V/V | SSE4_1 | Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 |
| 66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64 | 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 20 /r VPMOVSXBW xmm1, xmm2/m64 | 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 21 /r VPMOVSXBD xmm1, xmm2/m32 | 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 |

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| VEX.128.66.0F38 22 /r VPMOVSXBQ xmm1, xmm2/m16 | 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 23 /r VPMOVSXWD xmm1, xmm2/m64 | 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 24 /r VPMOVSXWQ xmm1, xmm2/m32 | 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 25 /r VPMOVSXDQ xmm1, xmm2/m64 | 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 |

Description

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

Packed_Sign_Extend_BYTE_to_WORD

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] \leftarrow SignExtend(SRC[55:48]);
 DEST[127:112] \leftarrow SignExtend(SRC[63:56]);

Packed_Sign_Extend_BYTE_to_DWORD
 DEST[31:0] \leftarrow SignExtend(SRC[7:0]);
 DEST[63:32] \leftarrow SignExtend(SRC[15:8]);
 DEST[95:64] \leftarrow SignExtend(SRC[23:16]);
 DEST[127:96] \leftarrow SignExtend(SRC[31:24]);

Packed_Sign_Extend_BYTE_to_QWORD
 DEST[63:0] \leftarrow SignExtend(SRC[7:0]);
 DEST[127:64] \leftarrow SignExtend(SRC[15:8]);

Packed_Sign_Extend_WORD_to_DWORD
 DEST[31:0] \leftarrow SignExtend(SRC[15:0]);
 DEST[63:32] \leftarrow SignExtend(SRC[31:16]);
 DEST[95:64] \leftarrow SignExtend(SRC[47:32]);
 DEST[127:96] \leftarrow SignExtend(SRC[63:48]);

Packed_Sign_Extend_WORD_to_QWORD
 DEST[63:0] \leftarrow SignExtend(SRC[15:0]);
 DEST[127:64] \leftarrow SignExtend(SRC[31:16]);

Packed_Sign_Extend_DWORD_to_QWORD
 DEST[63:0] \leftarrow SignExtend(SRC[31:0]);
 DEST[127:64] \leftarrow SignExtend(SRC[63:32]);

VPMOVSXBW

Packed_Sign_Extend_BYTE_to_WORD()
 DEST[255:128] \leftarrow 0

VPMOVSXBD

Packed_Sign_Extend_BYTE_to_DWORD()
 DEST[255:128] \leftarrow 0

VPMOVSXBQ

Packed_Sign_Extend_BYTE_to_QWORD()
 DEST[255:128] \leftarrow 0

VPMOVSXWD

Packed_Sign_Extend_WORD_to_DWORD()
 DEST[255:128] \leftarrow 0

INSTRUCTION SET REFERENCE

VPMOVSXWQ

Packed_Sign_Extend_WORD_to_QWORD()

DEST[255:128] ← 0

VPMOVSXDQ

Packed_Sign_Extend_DWORD_to_QWORD()

DEST[255:128] ← 0

PMOVSXBW

Packed_Sign_Extend_BYTE_to_WORD()

DEST[255:128] (Unmodified)

PMOVSXBD

Packed_Sign_Extend_BYTE_to_DWORD()

DEST[255:128] (Unmodified)

PMOVSXBQ

Packed_Sign_Extend_BYTE_to_QWORD()

DEST[255:128] (Unmodified)

PMOVSXWD

Packed_Sign_Extend_WORD_to_DWORD()

DEST[255:128] (Unmodified)

PMOVSXWQ

Packed_Sign_Extend_WORD_to_QWORD()

DEST[255:128] (Unmodified)

PMOVSXDQ

Packed_Sign_Extend_DWORD_to_QWORD()

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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_ cvtupi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

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

PMOVZX - Packed Move with Zero Extend

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64 | 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 PMOVZXBW xmm1, xmm2/m32 | V/V | SSE4_1 | Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 |
| 66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16 | V/V | SSE4_1 | Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 |
| 66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64 | V/V | SSE4_1 | Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 |
| 66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32 | V/V | SSE4_1 | Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 |
| 66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64 | 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 30 /r VPMOVZXBW xmm1, xmm2/m64 | 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 31 /r VPMOVZXBW xmm1, xmm2/m32 | 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 |

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| VEX.128.66.0F38 32 /r VPMOVZXBQ xmm1, xmm2/m16 | 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 33 /r VPMOVZXWD xmm1, xmm2/m64 | 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 34 /r VPMOVZXWQ xmm1, xmm2/m32 | V/V | AVX | Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 |
| VEX.128.66.0F38 35 /r VPMOVZXDQ xmm1, xmm2/m64 | 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 |

Description

Packed byte, word, or dword integers in 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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

Packed_Zero_Extend_BYTE_to_WORD

```

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]);

```

INSTRUCTION SET REFERENCE

DEST[127:112] \leftarrow ZeroExtend(SRC[63:56]);

Packed_Zero_Extend_BYTE_to_DWORD

DEST[31:0] \leftarrow ZeroExtend(SRC[7:0]);

DEST[63:32] \leftarrow ZeroExtend(SRC[15:8]);

DEST[95:64] \leftarrow ZeroExtend(SRC[23:16]);

DEST[127:96] \leftarrow ZeroExtend(SRC[31:24]);

Packed_Zero_Extend_BYTE_to_QWORD

DEST[63:0] \leftarrow ZeroExtend(SRC[7:0]);

DEST[127:64] \leftarrow ZeroExtend(SRC[15:8]);

Packed_Zero_Extend_WORD_to_DWORD

DEST[31:0] \leftarrow ZeroExtend(SRC[15:0]);

DEST[63:32] \leftarrow ZeroExtend(SRC[31:16]);

DEST[95:64] \leftarrow ZeroExtend(SRC[47:32]);

DEST[127:96] \leftarrow ZeroExtend(SRC[63:48]);

Packed_Zero_Extend_WORD_to_QWORD

DEST[63:0] \leftarrow ZeroExtend(SRC[15:0]);

DEST[127:64] \leftarrow ZeroExtend(SRC[31:16]);

Packed_Zero_Extend_DWORD_to_QWORD

DEST[63:0] \leftarrow ZeroExtend(SRC[31:0]);

DEST[127:64] \leftarrow ZeroExtend(SRC[63:32]);

VPMOVZXBW

Packed_Zero_Extend_BYTE_to_WORD()

DEST[255:128] \leftarrow 0

VPMOVZXB D

Packed_Zero_Extend_BYTE_to_DWORD()

DEST[255:128] \leftarrow 0

VPMOVZXBQ

Packed_Zero_Extend_BYTE_to_QWORD()

DEST[255:128] \leftarrow 0

VPMOVZXWD

Packed_Zero_Extend_WORD_to_DWORD()

DEST[255:128] \leftarrow 0

VPMOVZXWQ

Packed_Zero_Extend_WORD_to_QWORD()
 DEST[255:128] ← 0

VPMOVZXDQ

Packed_Zero_Extend_DWORD_to_QWORD()
 DEST[255:128] ← 0

PMOVZXBW

Packed_Zero_Extend_BYTE_to_WORD()
 DEST[255:128] (Unmodified)

PMOVZXB

Packed_Zero_Extend_BYTE_to_DWORD()
 DEST[255:128] (Unmodified)

PMOVZXBQ

Packed_Zero_Extend_BYTE_to_QWORD()
 DEST[255:128] (Unmodified)

PMOVZXWD

Packed_Zero_Extend_WORD_to_DWORD()
 DEST[255:128] (Unmodified)

PMOVZXWQ

Packed_Zero_Extend_WORD_to_QWORD()
 DEST[255:128] (Unmodified)

PMOVZXDQ

Packed_Zero_Extend_DWORD_to_QWORD()
 DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMOVZXBW __m128i _mm_cvtepu8_epi16 (__m128i a);

PMOVZXB __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);

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD

If VEX.L = 1.

If VEX.vvvv != 1111B.

PMULHUW - Multiply Packed Unsigned Integers and Store High Result

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F E4 /r PMULHUW xmm1, xmm2/m128 | 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 VPMULHUW xmm1, xmm2, xmm3/m128 | 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. |

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.

The second source operand is an XMM register or a 128-bit memory location. The destination operand and first source operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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]

```

INSTRUCTION SET REFERENCE

DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULHUW __m128i _mm_mulhi_epu16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULHRSW - Multiply Packed Unsigned Integers with Round and Shift

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128 | 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 VPMULHRSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1. |

Description

PMULHRSW 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. The first source and destination operands are XMM registers. The second source operand is an XMM register or 128-bit memory location.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
```

INSTRUCTION SET REFERENCE

DEST[63:48] \leftarrow temp3[16:1]
DEST[79:64] \leftarrow temp4[16:1]
DEST[95:80] \leftarrow temp5[16:1]
DEST[111:96] \leftarrow temp6[16:1]
DEST[127:112] \leftarrow temp7[16:1]
DEST[255:128] \leftarrow 0

PMULHRSW (128-bit Legacy SSE version)

temp0[31:0] \leftarrow INT32 ((DEST[15:0] * SRC[15:0]) \gg 14) + 1
temp1[31:0] \leftarrow INT32 ((DEST[31:16] * SRC[31:16]) \gg 14) + 1
temp2[31:0] \leftarrow INT32 ((DEST[47:32] * SRC[47:32]) \gg 14) + 1
temp3[31:0] \leftarrow INT32 ((DEST[63:48] * SRC[63:48]) \gg 14) + 1
temp4[31:0] \leftarrow INT32 ((DEST[79:64] * SRC[79:64]) \gg 14) + 1
temp5[31:0] \leftarrow INT32 ((DEST[95:80] * SRC[95:80]) \gg 14) + 1
temp6[31:0] \leftarrow INT32 ((DEST[111:96] * SRC[111:96]) \gg 14) + 1
temp7[31:0] \leftarrow INT32 ((DEST[127:112] * SRC[127:112]) \gg 14) + 1
DEST[15:0] \leftarrow temp0[16:1]
DEST[31:16] \leftarrow temp1[16:1]
DEST[47:32] \leftarrow temp2[16:1]
DEST[63:48] \leftarrow temp3[16:1]
DEST[79:64] \leftarrow temp4[16:1]
DEST[95:80] \leftarrow temp5[16:1]
DEST[111:96] \leftarrow temp6[16:1]
DEST[127:112] \leftarrow temp7[16:1]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULHRSW __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULHW - Multiply Packed Integers and Store High Result

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F E5 /r PMULHW xmm1, xmm2/m128 | 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 VPMULHW xmm1, xmm2, xmm3/m128 | 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. |

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. The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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]

INSTRUCTION SET REFERENCE

DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULHW __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULLW/PMULLD - Multiply Packed Integers and Store Low Result

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F D5 /r PMULLW xmm1, xmm2/m128 | 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. |
| 66 0F 38 40 /r PMULLD xmm1, xmm2/m128 | 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.128.66.0F D5 /r VPMULLW xmm1, xmm2, xmm3/m128 | 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.128.66.0F38 40 /r VPMULLD xmm1, xmm2, xmm3/m128 | 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 |

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(32) bits of each intermediate 32-bit(64-bit) result in the destination operand. (Figure 4-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, shows this operation when using 64-bit operands.) The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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]

INSTRUCTION SET REFERENCE

Temp3[63:0] \leftarrow SRC1[127:96] * SRC2[127:96]
DEST[31:0] \leftarrow Temp0[31:0]
DEST[63:32] \leftarrow Temp1[31:0]
DEST[95:64] \leftarrow Temp2[31:0]
DEST[127:96] \leftarrow Temp3[31:0]
DEST[255:128] \leftarrow 0

PMULLD (128-bit Legacy SSE version)

Temp0[63:0] \leftarrow DEST[31:0] * SRC[31:0]
Temp1[63:0] \leftarrow DEST[63:32] * SRC[63:32]
Temp2[63:0] \leftarrow DEST[95:64] * SRC[95:64]
Temp3[63:0] \leftarrow DEST[127:96] * SRC[127:96]
DEST[31:0] \leftarrow Temp0[31:0]
DEST[63:32] \leftarrow Temp1[31:0]
DEST[95:64] \leftarrow Temp2[31:0]
DEST[127:96] \leftarrow Temp3[31:0]
DEST[255:128] (Unmodified)

VPMULLW (VEX.128 encoded version)

Temp0[31:0] \leftarrow SRC1[15:0] * SRC2[15:0]
Temp1[31:0] \leftarrow SRC1[31:16] * SRC2[31:16]
Temp2[31:0] \leftarrow SRC1[47:32] * SRC2[47:32]
Temp3[31:0] \leftarrow SRC1[63:48] * SRC2[63:48]
Temp4[31:0] \leftarrow SRC1[79:64] * SRC2[79:64]
Temp5[31:0] \leftarrow SRC1[95:80] * SRC2[95:80]
Temp6[31:0] \leftarrow SRC1[111:96] * SRC2[111:96]
Temp7[31:0] \leftarrow SRC1[127:112] * SRC2[127:112]
DEST[15:0] \leftarrow Temp0[15:0]
DEST[31:16] \leftarrow Temp1[15:0]
DEST[47:32] \leftarrow Temp2[15:0]
DEST[63:48] \leftarrow Temp3[15:0]
DEST[79:64] \leftarrow Temp4[15:0]
DEST[95:80] \leftarrow Temp5[15:0]
DEST[111:96] \leftarrow Temp6[15:0]
DEST[127:112] \leftarrow Temp7[15:0]
DEST[255:128] \leftarrow 0

PMULLW (128-bit Legacy SSE version)

Temp0[31:0] \leftarrow DEST[15:0] * SRC[15:0]
Temp1[31:0] \leftarrow DEST[31:16] * SRC[31:16]
Temp2[31:0] \leftarrow DEST[47:32] * SRC[47:32]
Temp3[31:0] \leftarrow DEST[63:48] * SRC[63:48]
Temp4[31:0] \leftarrow 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULLW __m128i _mm_mullo_epi16 (__m128i a, __m128i b)

PMULLUD __m128i _mm_mullo_epi32(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULUDQ - Multiply Packed Unsigned Doubleword Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F F4 /r PMULUDQ xmm1, xmm2/m128 | V/V | SSE2 | Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1. |
| VEX.NDS.128.66.0F F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128 | 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. |

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand. The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The first source operand is two packed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULUDQ (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$

PMULUDQ (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0]$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[95:64] * \text{SRC}[95:64]$$

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ __m128i _mm_mul_epu32 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PMULDQ - Multiply Packed Doubleword Integers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 28 /r PMULDQ xmm1, xmm2/m128 | 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 28 /r VPMULDQ xmm1, xmm2, xmm3/m128 | 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. |

Description

Multiplies the first source operand by the second source operand and stores the result in the destination operand. The second source operand is two packed signed doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The first source operand is two packed signed doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed signed quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPMULDQ (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$

PMULDQ (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0]$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[95:64] * \text{SRC}[95:64]$$

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

POR - Bitwise Logical Or

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|-----------------------------------|
| 66 0F EB /r POR xmm1, xmm2/m128 | V/V | SSE2 | Bitwise OR of xmm2/m128 and xmm1. |
| VEX.NDS.128.66.0F EB /r VPOR xmm1, xmm2, xmm3/m128 | V/V | AVX | Bitwise OR of xmm2/m128 and xmm3. |

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. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPOR (VEX.128 encoded version)

DEST ← SRC1 OR SRC2

DEST[255:128] ← 0

POR (128-bit Legacy SSE version)

DEST ← DEST OR SRC

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

POR __m128i _mm_or_si128 (__m128i a, __m128i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD

If VEX.LI = 1.

PSADBW - Compute Sum of Absolute Differences

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F F6 /r PSADBW xmm1, xmm2/m128 | 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 VPSADBW xmm1, xmm2, xmm3/m128 | 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. |

Description

Computes the absolute value of the difference of packed groups of 8 unsigned byte integers from the second 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 bit 79:64 of the destination. The remaining words of the destination are set to 0.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPSADBW (VEX.128 encoded version)

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(* Repeat operation for bytes 2 through 14 *)

TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)

DEST[127:80] ← 000000000000

DEST[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSHUFB - Packed Shuffle Bytes

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 00 /r PSHUFB xmm1, xmm2/m128 | V/V | SSSE3 | Shuffle bytes in xmm1 according to contents of xmm2/m128. |
| VEX.NDS.128.66.0F38 00 /r VPSHUFB xmm1, xmm2, xmm3/m128 | V/V | AVX | Shuffle bytes in xmm2 according to contents of xmm3/m128. |

Description

Shuffles bytes in the first source operand according to the shuffle control mask in the second source operand. The instruction permutes byte 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 element in the shuffle control mask provides an index field to select the byte element in the first source operand. The index field is defined as the least significant 4 bits of each byte element of the shuffle control mask. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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[255:128] ← 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[255:128] (Unmodified)

```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFB __m128i _mm_shuffle_epi8(__m128i a, __m128i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSHUFD - Shuffle Packed Doublewords

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8 | 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 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8 | V/V | AVX | Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1. |

Description

Copies doublewords from source operand and inserts them in the destination operand at the locations selected with the immediate control operand. Figure 5-29 shows the operation of the PSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location in the destination operand. For example, bits 0 and 1 of the order operand select the contents of doubleword 0 of the destination operand. The encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 5-29) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

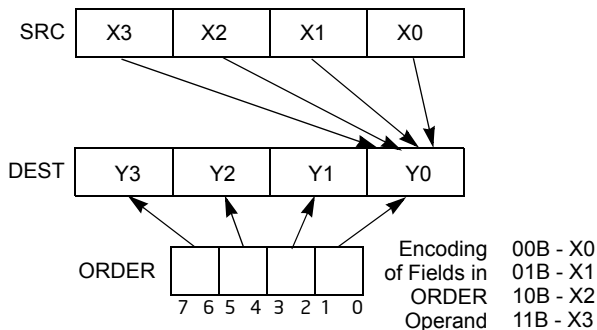


Figure 5-29. PSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note

that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[255:128] ← 0
```

PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFD `__m128i _mm_shuffle_epi32(__m128i a, int n)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

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

PSHUFHW - Shuffle Packed High Words

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8 | 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 VPSHUFHW xmm1, xmm2/m128, imm8 | V/V | AVX | Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1. |

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 4-7 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. 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.

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

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

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

PSHUFLW - Shuffle Packed Low Words

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8 | 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 VPSHUFLW xmm1, xmm2/m128, imm8 | V/V | AVX | Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1. |

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 5-29. 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.

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

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSHUFLW (VEX.128 encoded version)

DEST[15:0] ← (SRC1 >> (imm[1:0] * 16))[15:0]

DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]

DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]

DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]

DEST[127:64] ← SRC[127:64]

DEST[255:128] ← 0

PSHUFLW (128-bit Legacy SSE version)

$\text{DEST}[15:0] \leftarrow (\text{SRC} \gg (\text{imm}[1:0] * 16))[15:0]$
 $\text{DEST}[31:16] \leftarrow (\text{SRC} \gg (\text{imm}[3:2] * 16))[15:0]$
 $\text{DEST}[47:32] \leftarrow (\text{SRC} \gg (\text{imm}[5:4] * 16))[15:0]$
 $\text{DEST}[63:48] \leftarrow (\text{SRC} \gg (\text{imm}[7:6] * 16))[15:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC}[127:64]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFLW `__m128i _mm_shufflelo_epi16(__m128i a, int n)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

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

PSIGNB/PSIGNW/PSIGND - Packed SIGN

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 66 0F 38 08 /r PSIGNB xmm1, xmm2/m128 | V/V | SSSE3 | Negate/zero/preserve packed byte integers in xmm1 depending on the corresponding sign in xmm2/m128. |
| 66 0F 38 09 /r PSIGNW xmm1, xmm2/m128 | V/V | SSSE3 | Negate/zero/preserve packed word integers in xmm1 depending on the corresponding sign in xmm2/m128. |
| 66 0F 38 0A /r PSIGND xmm1, xmm2/m128 | V/V | SSSE3 | Negate/zero/preserve packed doubleword integers in xmm1 depending on the corresponding sign in xmm2/m128. |
| VEX.NDS.128.66.0F38 08 /r VPSIGNB xmm1, xmm2, xmm3/m128 | V/V | AVX | Negate/zero/preserve packed byte integers in xmm2 depending on the corresponding sign in xmm3/m128. |
| VEX.NDS.128.66.0F38 09 /r VPSIGNW xmm1, xmm2, xmm3/m128 | V/V | AVX | Negate/zero/preserve packed word integers in xmm2 depending on the corresponding sign in xmm3/m128. |
| VEX.NDS.128.66.0F38 0A /r VPSIGND xmm1, xmm2, xmm3/m128 | V/V | AVX | Negate/zero/preserve packed doubleword integers in xmm2 depending on the corresponding sign in xmm3/m128. |

Description

PSIGNB/PSIGNW/PSIGND negates each data element of the first source operand if the signed integer value of the corresponding data element in the second source operand is less than zero. If the signed integer value of a data element in the second source operand is positive, the corresponding data element in the first source operand is unchanged. If a data element in the second source operand is zero, the corresponding data element in the first source operand is set to zero.

PSIGNB operates on signed bytes. PSIGNW operates on 16-bit signed words. PSIGND operates on signed 32-bit integers.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

BYTE_SIGN(SRC1, SRC2)

```
if (SRC2[7..0] < 0 )
    DEST[7...0] ← Neg(SRC1[7...0])
else if(SRC2[7..0] == 0 )
    DEST[7...0] ← 0
else if(SRC2[7..0] > 0 )
    DEST[7...0] ← SRC1[7...0]
```

Repeat operation for 2nd through 15th bytes

```
if (SRC2[127..120] < 0 )
    DEST[127...120] ← Neg(SRC1[127...120])
else if(SRC2[127.. 120] == 0 )
    DEST[127...120] ← 0
else if(SRC2[127.. 120] > 0 )
    DEST[127...120] ← SRC1[127...120]
```

WORD_SIGN(SRC1, SRC2)

```
if (SRC2[15..0] < 0 )
    DEST[15...0] ← Neg(SRC1[15...0])
else if(SRC2[15..0] == 0 )
    DEST[15...0] ← 0
else if(SRC2[15..0] > 0 )
    DEST[15...0] ← SRC1[15...0]
```

Repeat operation for 2nd through 7th words

```
if (SRC2[127..112] < 0 )
    DEST[127...112] ← Neg(SRC1[127...112])
else if(SRC2[127.. 112] == 0 )
    DEST[127...112] ← 0
else if(SRC2[127.. 112] > 0 )
    DEST[127...112] ← SRC1[127...112]
```

DWORD_SIGN(SRC1, SRC2)

```
if (SRC2[31..0] < 0 )
    DEST[31...0] ← Neg(SRC1[31...0])
else if(SRC2[31..0] == 0 )
    DEST[31...0] ← 0
else if(SRC2[31..0] > 0 )
    DEST[31...0] ← SRC1[31...0]
```

INSTRUCTION SET REFERENCE

Repeat operation for 2nd through 3rd double words

```
if (SRC2[127..96] < 0 )
    DEST[127...96] ← Neg(SRC1[127...96])
else if(SRC2[127.. 96] == 0 )
    DEST[127...96] ← 0
else if(SRC2[127.. 96] > 0 )
    DEST[127...96] ← SRC1[127...96]
```

VPSIGNB (VEX.128 encoded version)

```
DEST[127:0] ← BYTE_SIGN(SRC1, SRC2)
DEST[255:128] ← 0
```

PSIGNB (128-bit Legacy SSE version)

```
DEST[127:0] ← BYTE_SIGN(DEST, SRC)
DEST[255:128] (Unmodified)
```

VPSIGNW (VEX.128 encoded version)

```
DEST[127:0] ← WORD_SIGN(SRC1, SRC2)
DEST[255:128] ← 0
```

PSIGNW (128-bit Legacy SSE version)

```
DEST[127:0] ← WORD_SIGN(DEST, SRC)
DEST[255:128] (Unmodified)
```

VPSIGND (VEX.128 encoded version)

```
DEST[127:0] ← DWORD_SIGN(SRC1, SRC2)
DEST[255:128] ← 0
```

PSIGND (128-bit Legacy SSE version)

```
DEST[127:0] ← DWORD_SIGN(DEST, SRC)
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSIGNB __m128i _mm_sign_epi8 (__m128i a, __m128i b)
```

```
PSIGNW __m128i _mm_sign_epi16 (__m128i a, __m128i b)
```

```
PSIGND __m128i _mm_sign_epi32 (__m128i a, __m128i b)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PSLLDQ - Byte Shift Left

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 73 /7 ib PSLLDQ xmm1, imm8 | V/V | SSE2 | Shift xmm1 left by imm8 bytes while shifting in 0s. |
| VEX.NDD.128.66.0F 73 /7 ib VPSLLDQ xmm1, xmm2, imm8 | V/V | AVX | Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1. |

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSLLDQ (VEX.128 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← SRC << (TEMP * 8)

DEST[255:128] ← 0

PSLLDQ(128-bit Legacy SSE version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← DEST << (TEMP * 8)

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSLLDQ __m128i _mm_slli_si128 (__m128i a, int imm)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.L = 1.

PSRLDQ - Byte Shift Right

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 73 /3 ib PSRLDQ xmm1, imm8 | V/V | SSE2 | Shift xmm1 right by imm8 bytes while shifting in 0s. |
| VEX.NDD.128.66.0F 73 /3 ib VPSRLDQ xmm1, xmm2, imm8 | V/V | AVX | Shift xmm1 right by imm8 bytes while shifting in 0s. |

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 (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSRLDQ (VEX.128 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← SRC >> (TEMP * 8)

DEST[255:128] ← 0

PSRLDQ(128-bit Legacy SSE version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST ← DEST >> (TEMP * 8)

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSRLDQ __m128i _mm_srli_si128 (__m128i a, int imm)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.L = 1.

PSLLW/PSLLD/PSLLQ - Bit Shift Left

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|---------------------------------------|-----------------------------------|---|
| 66 0F F1/r PSLLW xmm1, xmm2/m128 | V/V | SSE2 | Shift words in xmm1 left by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 71 /6 ib PSLLW xmm1, imm8 | V/V | SSE2 | Shift words in xmm1 left by imm8 while shifting in 0s. |
| 66 0F F2 /r PSLLD xmm1, xmm2/m128 | V/V | SSE2 | Shift doublewords in xmm1 left by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 72 /6 ib PSLLD xmm1, imm8 | V/V | SSE2 | Shift doublewords in xmm1 left by imm8 while shifting in 0s. |
| 66 0F F3 /r PSLLQ xmm1, xmm2/m128 | V/V | SSE2 | Shift quadwords in xmm1 left by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 73 /6 ib PSLLQ xmm1, imm8 | V/V | SSE2 | Shift quadwords in xmm1 left by imm8 while shifting in 0s. |
| VEX.NDS.128.66.0F F1 /r VPSLLW xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.NDD.128.66.0F 71 /6 ib VPSLLW xmm1, xmm2, imm8 | V/V | AVX | Shift words in xmm2 left by imm8 while shifting in 0s. |
| VEX.NDS.128.66.0F F2 /r VPSLLD xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.NDD.128.66.0F 72 /6 ib VPSLLD xmm1, xmm2, imm8 | V/V | AVX | Shift doublewords in xmm2 left by imm8 while shifting in 0s. |

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| VEX.NDS.128.66.0F F3 /r VPSLLQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.NDD.128.66.0F 73 /6 ib VPSLLQ xmm1, xmm2, imm8 | V/V | AVX | Shift quadwords in xmm2 left by imm8 while shifting in 0s. |

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.

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

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.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. For shifts with an immediate count (VEX.128.66.0F 71-73 /6), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

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);

INSTRUCTION SET REFERENCE

(* Repeat shift operation for 2nd through 7th words *)

DEST[127:112] ← ZeroExtend(SRC[127:112] << 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_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;

VPSLLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)

DEST[255:128] ← 0

VPSLLW (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)

DEST[255:128] ← 0

PSLLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)

DEST[255:128] (Unmodified)

PSLLW (xmm, imm8)

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

DEST[255:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)

DEST[255:128] ← 0

VPSLLD (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)

DEST[255:128] ← 0

PSLLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)

DEST[255:128] (Unmodified)

PSLLD (xmm, imm8)

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

DEST[255:128] (Unmodified)

VPSLLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)

DEST[255:128] ← 0

VPSLLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)

DEST[255:128] ← 0

PSLLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)

DEST[255:128] (Unmodified)

PSLLQ (xmm, imm8)

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

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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)

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

None

Other Exceptions

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

#UD If VEX.L I= 1.

PSRAW/PSRAD - Bit Shift Arithmetic Right

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F E1/r PSRAW xmm1, xmm2/m128 | V/V | SSE2 | Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits. |
| 66 0F 71 /4 ib PSRAW xmm1, imm8 | V/V | SSE2 | Shift words in xmm1 right by imm8 while shifting in sign bits. |
| 66 0F E2 /r PSRAD xmm1, xmm2/m128 | V/V | SSE2 | Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in sign bits. |
| 66 0F 72 /4 ib PSRAD xmm1, imm8 | V/V | SSE2 | Shift doublewords in xmm1 right by imm8 while shifting in sign bits. |
| VEX.NDS.128.66.0F E1 /r VPSRAW xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits. |
| VEX.NDD.128.66.0F 71 /4 ib VPSRAW xmm1, xmm2, imm8 | V/V | AVX | Shift words in xmm2 right by imm8 while shifting in sign bits. |
| VEX.NDS.128.66.0F E2 /r VPSRAD xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits. |
| VEX.NDD.128.66.0F 72 /4 ib VPSRAD xmm1, xmm2, imm8 | V/V | AVX | Shift doublewords in xmm2 right by imm8 while shifting in sign bits. |

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 left, 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.

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

The 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; the PSRAD instruction shifts each of the doublewords in the first source operand; and the PSRAQ instruction shifts the quadword (or quadwords) in the first source operand.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. : Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. For shifts with an immediate count (VEX.128.66.0F 71-73 /4), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD. : Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

Operation

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 (xmm, xmm, xmm/m128)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
 DEST[255:128] \leftarrow 0

VPSRAW (xmm, imm8)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
 DEST[255:128] \leftarrow 0

PSRAW (xmm, xmm, xmm/m128)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
 DEST[255:128] (Unmodified)

PSRAW (xmm, imm8)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
 DEST[255:128] (Unmodified)

VPSRAD (xmm, xmm, xmm/m128)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
 DEST[255:128] \leftarrow 0

VPSRAD (xmm, imm8)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
 DEST[255:128] \leftarrow 0

PSRAD (xmm, xmm, xmm/m128)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
 DEST[255:128] (Unmodified)

PSRAD (xmm, imm8)

DEST[127:0] \leftarrow ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
 DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSRAW `__m128i _mm_srai_epi16 (__m128i m, int count)`

PSRAW `__m128i _mm_sra_epi16 (__m128i m, __m128i count)`

PSRAD `__m128i _mm_srai_epi32 (__m128i m, int count)`

PSRAD `__m128i _mm_sra_epi32 (__m128i m, __m128i count)`

SIMD Floating-Point Exceptions

None

INSTRUCTION SET REFERENCE

Other Exceptions

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

#UD If VEX.L = 1.

PSRLW/PSRLD/PSRLQ - Shift Packed Data Right Logical

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F D1 /r PSRLW xmm1, xmm2/m128 | V/V | SSE2 | Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 71 /2 ib PSRLW xmm1, imm8 | V/V | SSE2 | Shift words in xmm1 right by imm8 while shifting in 0s. |
| 66 0F D2 /r PSRLD xmm1, xmm2/m128 | V/V | SSE2 | Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 72 /2 ib PSRLD xmm1, imm8 | V/V | SSE2 | Shift doublewords in xmm1 right by imm8 while shifting in 0s. |
| 66 0F D3 /r PSRLQ xmm1, xmm2/m128 | V/V | SSE2 | Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s. |
| 66 0F 73 /2 ib PSRLQ xmm1, imm8 | V/V | SSE2 | Shift quadwords in xmm1 right by imm8 while shifting in 0s. |
| VEX.NDS.128.66.0F D1 /r VPSRLW xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.NDD.128.66.0F 71 /2 ib VPSRLW xmm1, xmm2, imm8 | V/V | AVX | Shift words in xmm2 right by imm8 while shifting in 0s. |
| VEX.NDS.128.66.0F D2 /r VPSRLD xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.NDD.128.66.0F 72 /2 ib VPSRLD xmm1, xmm2, imm8 | V/V | AVX | Shift doublewords in xmm2 right by imm8 while shifting in 0s. |

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| VEX.NDS.128.66.0F D3 /r VPSRLQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.NDD.128.66.0F 73 /2 ib VPSRLQ xmm1, xmm2, imm8 | V/V | AVX | Shift quadwords in xmm2 right by imm8 while shifting in 0s. |

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. 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.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. For shifts with an immediate count (VEX.128.66.0F 71-73 /2), VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register. VEX.L must be 0, otherwise instructions will #UD. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

Operation

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(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(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 (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)

DEST[255:128] ← 0

VPSRLW (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)

DEST[255:128] ← 0

PSRLW (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)

DEST[255:128] (Unmodified)

PSRLW (xmm, imm8)

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

DEST[255:128] (Unmodified)

VPSRLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)

INSTRUCTION SET REFERENCE

DEST[255:128] ← 0

VPSRLD (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)

DEST[255:128] ← 0

PSRLD (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)

DEST[255:128] (Unmodified)

PSRLD (xmm, imm8)

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

DEST[255:128] (Unmodified)

VPSRLQ (xmm, xmm, xmm/m128)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)

DEST[255:128] ← 0

VPSRLQ (xmm, imm8)

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)

DEST[255:128] ← 0

PSRLQ (xmm, xmm, xmm/m128)

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

DEST[255:128] (Unmodified)

PSRLQ (xmm, imm8)

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

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)

PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)

PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)

PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)

PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)

PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

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

#UD If VEX.L = 1.

PTEST- Packed Bit Test

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 38 17 /r PTEST xmm1, xmm2/m128 | V/V | SSE4_1 | Set ZF and CF depending on bit-wise AND and ANDN of sources |
| VEX.128.66.0F38 17 /r VPTEST xmm1, xmm2/m128 | V/V | AVX | Set ZF and CF depending on bit-wise AND and ANDN of sources |
| VEX.256.66.0F38 17 /r VPTEST ymm1, ymm2/m256 | V/V | AVX | Set ZF and CF depending on bit-wise AND and ANDN of sources |
| VEX.128.66.0F38 0E /r VTESTPS xmm1, xmm2/m128 | V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources |
| VEX.256.66.0F38 0E /r VTESTPS ymm1, ymm2/m256 | V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources |
| VEX.128.66.0F38 0F /r VTESTPD xmm1, xmm2/m128 | V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources |
| VEX.256.66.0F38 0F /r VTESTPD ymm1, ymm2/m256 | V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources |

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the inverted source sign bits with the dest sign bits produces all zeros the CF is set else the CF is clear.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the inverted source sign bits with the dest sign bits produces all zeros the CF is set else the CF is clear.

The first source register is specified by the ModR/M *reg* field.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 256-bit memory location. The destination register is not modified.

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

Operation

VPTEST (VEX.256 encoded version)

```
IF (SRC[255:0] BITWISE AND DEST[255:0] == 0) THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] == 0) THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

PTEST (128-bit versions)

```
IF (SRC[127:0] BITWISE AND DEST[127:0] == 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] == 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPS (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
```

INSTRUCTION SET REFERENCE

```
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] =  
TEMP[255] = 0)  
    THEN ZF ← 1;  
    ELSE ZF ← 0;
```

```
TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]  
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] =  
TEMP[255] = 0)  
    THEN CF ← 1;  
    ELSE CF ← 0;  
DEST (unmodified)  
AF ← OF ← PF ← SF ← 0;
```

VTESTPD (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]  
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)  
    THEN ZF ← 1;  
    ELSE ZF ← 0;
```

```
TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]  
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)  
    THEN CF ← 1;  
    ELSE CF ← 0;  
DEST (unmodified)  
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPTEST

```
int _mm256_testz_si256 (__m256i s1, __m256i s2);  
int _mm256_testc_si256 (__m256i s1, __m256i s2);  
int _mm256_testnzc_si256 (__m256i s1, __m256i s2);  
int _mm_testz_si128 (__m128i s1, __m128i s2);  
int _mm_testc_si128 (__m128i s1, __m128i s2);  
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

VTESTPS

```

int _mm256_testz_ps (__m256 s1, __m256 s2);
int _mm256_testc_ps (__m256 s1, __m256 s2);
int _mm256_testnzc_ps (__m256 s1, __m128 s2);
int _mm_testz_ps (__m128 s1, __m128 s2);
int _mm_testc_ps (__m128 s1, __m128 s2);
int _mm_testnzc_ps (__m128 s1, __m128 s2);

```

VTESTPD

```

int _mm256_testz_pd (__m256d s1, __m256d s2);
int _mm256_testc_pd (__m256d s1, __m256d s2);
int _mm256_testnzc_pd (__m256d s1, __m256d s2);
int _mm_testz_pd (__m128d s1, __m128d s2);
int _mm_testc_pd (__m128d s1, __m128d s2);
int _mm_testnzc_pd (__m128d s1, __m128d s2);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111.

PSUBB/PSUBW/PSUBD/PSUBQ -Packed Integer Subtract

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F F8 /r PSUBB xmm1, xmm2/m128 | V/V | SSE2 | Subtract packed byte integers in xmm2/m128 from xmm1. |
| 66 0F F9 /r PSUBW xmm1, xmm2/m128 | V/V | SSE2 | Subtract packed word integers in xmm2/m128 from xmm1. |
| 66 0F FA /r PSUBD xmm1, xmm2/m128 | V/V | SSE2 | Subtract packed doubleword integers in xmm2/m128 from xmm1. |
| 66 0F FB/r PSUBQ xmm1, xmm2/m128 | V/V | SSE2 | Subtract packed quadword integers in xmm2/m128 from xmm1. |
| VEX.NDS.128.66.0F F8 /r VPSUBB xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract packed byte integers in xmm3/m128 from xmm2. |
| VEX.NDS.128.66.0F F9 /r VPSUBW xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract packed word integers in xmm3/m128 from xmm2. |
| VEX.NDS.128.66.0F FA /r VPSUBD xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract packed doubleword integers in xmm3/m128 from xmm2. |
| VEX.NDS.128.66.0F FB/r VPSUBQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract packed quadword integers in xmm3/m128 from xmm2. |

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. The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. 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 of the values operated on.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

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[255:128] ← 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]

INSTRUCTION SET REFERENCE

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[255:128] (Unmodified)

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[255:128] ← 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[255:128] (Unmodified)

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[255:128] ← 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[255:128] (Unmodified)

VPSUBQ (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0]-SRC2[63:0]

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

DEST[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1

PSUBSB/PSUBSW - Subtract Packed Signed Integers with Signed Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F E8 /r PSUBSB xmm1, xmm2/m128 | 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 PSUBSW xmm1, xmm2/m128 | 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 VPSUBSB xmm1, xmm2, xmm3/m128 | 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 VPSUBSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results. |

Description

Performs a SIMD subtract of the packed signed integers of the second source operand from the packed signed integers of the first source operand, and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The first source and destination operands are XMM registers and the second source operand is either an XMM register or a 128-bit memory location.

The 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 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: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSUBSB

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[255:128] ← 0

PSUBSB

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[255:128] (Unmodified)

VPSUBSW

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[255:128] ← 0

PSUBSW

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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSUBSB __m128i _mm_subs_epi8(__m128i m1, __m128i m2)

PSUBSW __m128i _mm_subs_epi16(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD if VEX.L = 1.

PSUBUSB/PSUBUSW -Subtract Packed Unsigned Integers with Unsigned Saturation

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F D8 /r PSUBUSB xmm1, xmm2/m128 | 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 PSUBUSW xmm1, xmm2/m128 | 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 VPSUBUSB xmm1, xmm2, xmm3/m128 | 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 VPSUBUSW xmm1, xmm2, xmm3/m128 | V/V | AVX | Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result. |

Description

Performs a SIMD subtract of the packed unsigned integers of the second source operand from the packed unsigned integers of the first source operand and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

The first source and destination operands are XMM registers. The second source operand can be either an XMM register or a 128-bit memory location.

The 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 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: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPSUBUSB

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[255:128] ← 0

PSUBUSB

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[255:128] (Unmodified)

VPSUBUSW

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[255:128] ← 0

PSUBUSW

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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PSUBUSB __m128i _mm_subs_epu8(__m128i m1, __m128i m2)
 PSUBUSW __m128i _mm_subs_epu16(__m128i m1, __m128i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

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

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ - Unpack High Data

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 68/r PUNPCKHBW xmm1,xmm2/m128 | V/V | SSE2 | Interleave high-order bytes from xmm1 and xmm2/m128 into xmm1. |
| 66 0F 69/r PUNPCKHWD xmm1,xmm2/m128 | V/V | SSE2 | Interleave high-order words from xmm1 and xmm2/m128 into xmm1. |
| 66 0F 6A/r PUNPCKHDQ xmm1, xmm2/m128 | V/V | SSE2 | Interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1. |
| 66 0F 6D/r PUNPCKHQDQ xmm1, xmm2/m128 | V/V | SSE2 | Interleave high-order quadword from xmm1 and xmm2/m128 into xmm1 register. |
| VEX.NDS.128.66.0F 68/r VPUNPCKHBW xmm1,xmm2, xmm3/m128 | V/V | AVX | Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1. |
| VEX.NDS.128.66.0F 69/r VPUNPCKHWD xmm1,xmm2, xmm3/m128 | V/V | AVX | Interleave high-order words from xmm2 and xmm3/m128 into xmm1. |
| VEX.NDS.128.66.0F 6A/r VPUNPCKHDQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1. |
| VEX.NDS.128.66.0F 6D/r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Interleave high-order quadword from xmm2 and xmm3/m128 into xmm1 register. |

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 5-30 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

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

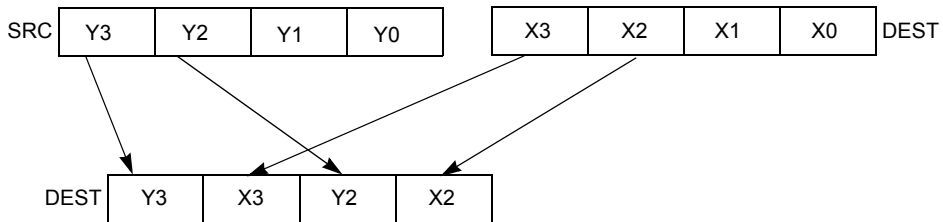


Figure 5-30. PUNPCKHDQ Instruction Operation

The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The 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 versions: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)

```

DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC2[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]

```

INSTRUCTION SET REFERENCE

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 (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(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(SRC1, SRC2)

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW

DEST[127:0] ← INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHBW

DEST[127:0] ← INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKHWD

DEST[127:0] ← INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHWD

DEST[127:0] ← INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKHDQ

DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)

DEST[255:127] (Unmodified)

VPUNPCKHDQ

DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)

DEST[255:127] ← 0

PUNPCKHQDQ

DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(DEST, SRC)

DEST[255:127] (Unmodified)

VPUNPCKHQDQ

DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)

DEST[255:127] ← 0

Intel C/C++ Compiler Intrinsic EquivalentPUNPCKHBW `__m128i_mm_unpackhi_epi8(__m128i m1, __m128i m2)`PUNPCKHWD `__m128i_mm_unpackhi_epi16(__m128i m1, __m128i m2)`PUNPCKHDQ `__m128i_mm_unpackhi_epi32(__m128i m1, __m128i m2)`PUNPCKHQDQ `__m128i_mm_unpackhi_epi64 (__m128i a, __m128i b)`**SIMD Floating-Point Exceptions**

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ - Unpack Low Data

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 60/r PUNPCKLBW xmm1,xmm2/m128 | V/V | SSE2 | Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1. |
| 66 0F 61/r PUNPCKLWD xmm1,xmm2/m128 | V/V | SSE2 | Interleave low-order words from xmm1 and xmm2/m128 into xmm1. |
| 66 0F 62/r PUNPCKLDQ xmm1, xmm2/m128 | V/V | SSE2 | Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1. |
| 66 0F 6C/r PUNPCKLQDQ xmm1, xmm2/m128 | V/V | SSE2 | Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register. |
| VEX.NDS.128.66.0F 60/r VPUNPCKLBW xmm1,xmm2, xmm3/m128 | V/V | AVX | Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1. |
| VEX.NDS.128.66.0F 61/r VPUNPCKLWD xmm1,xmm2, xmm3/m128 | V/V | AVX | Interleave low-order words from xmm2 and xmm3/m128 into xmm1. |
| VEX.NDS.128.66.0F 62/r VPUNPCKLDQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1. |
| VEX.NDS.128.66.0F 6C/r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128 | V/V | AVX | Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register. |

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 5-31 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

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

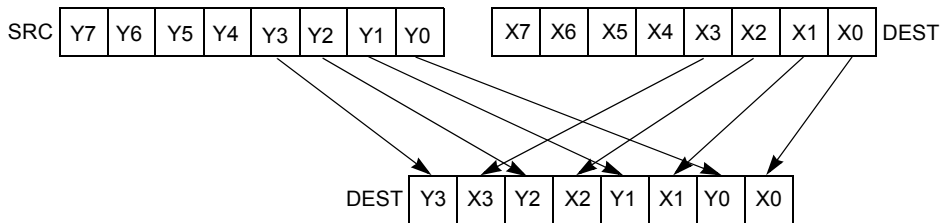


Figure 5-31. PUNPCKLBW Instruction Operation using 64-bit Operands

The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The 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.

128-bit Legacy SSE versions: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

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]

```

INSTRUCTION SET REFERENCE

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 (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(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(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

DEST[127:0] ← INTERLEAVE_BYTES(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKLWD

DEST[127:0] ← INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLWD

DEST[127:0] ← INTERLEAVE_WORDS(SRC1, SRC2)
DEST[255:127] ← 0

PUNPCKLDQ

DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)

DEST[255:127] (Unmodified)

VPUNPCKLDQ

DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)

DEST[255:127] ← 0

PUNPCKLQDQ

DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)

DEST[255:127] (Unmodified)

VPUNPCKLQDQ

DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)

DEST[255:127] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PUNPCKLBW `__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)`

PUNPCKLWD `__m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)`

PUNPCKLDQ `__m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)`

PUNPCKLQDQ `__m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

PXOR - Exclusive Or

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|------------------------------------|
| 66 0F EF /r PXOR xmm1, xmm2/m128 | V/V | SSE2 | Bitwise XOR of xmm2/m128 and xmm1. |
| VEX.NDS.128.66.0F EF /r VPXOR xmm1, xmm2, xmm3/m128 | V/V | AVX | Bitwise XOR of xmm3/m128 and xmm2. |

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. The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are different; otherwise, each bit is 0 if the corresponding bits of the first and second operand are the same.

128-bit Legacy SSE version: Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

VPXOR (VEX.128 encoded version)

DEST ← SRC1 XOR SRC2

DEST[255:128] ← 0

PXOR (128-bit Legacy SSE version)

DEST ← DEST XOR SRC

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

PXOR `__m128i _mm_xor_si128 (__m128i a, __m128i b)`

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4; additionally

#UD

If VEX.L = 1.

RCPPS- Compute Approximate Reciprocals of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 0F 53 /r RCPPS xmm1, xmm2/m128 | V/V | SSE | Computes the approximate reciprocals of packed single-precision values in xmm2/mem and stores the results in xmm1 |
| VEX.128.0F 53 /r VRCPPS xmm1, xmm2/m128 | V/V | AVX | Computes the approximate reciprocals of packed single-precision values in xmm2/mem and stores the results in xmm1 |
| VEX.256.0F 53 /r VRCPPS ymm1, ymm2/m256 | V/V | AVX | Computes the approximate reciprocals of packed single-precision values in ymm2/mem and stores the results in ymm1 |

Description

Performs an SIMD computation of the approximate reciprocals of the four or eight packed single precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1 for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| < 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register.

When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

Tiny results are always flushed to 0.0, with the sign of the operand:

- The result is guaranteed not to be tiny for inputs that are not greater than $(2^{125}) * (2^{-3} * 2^{-10})$ in absolute value.
- The result is guaranteed to be flushed to 0 for values greater than $(2^{126}) * (1 + 3 * 2^{-11})$ in absolute value.
- Input values in between this range may or may not produce tiny results,

depending on the implementation.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

VRCPPS (VEX.256 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
 DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
 DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
 DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
 DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
 DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
 DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
 DEST[255:224] ← APPROXIMATE(1/SRC[255:224])

VRCPPS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
 DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
 DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
 DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
 DEST[255:128] ← 0

RCPPS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
 DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
 DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
 DEST[127:96] ← APPROXIMATE(1/SRC[127:96])

INSTRUCTION SET REFERENCE

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

RCPPS __m256 _mm256_rcp_ps (__m256 a);

RCPPS __m128 _mm_rcp_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

RCPSS - Compute Reciprocal of Scalar Single-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| F3 0F 53 /r RCPSS xmm1, xmm2/m32 | V/V | SSE | Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm2/m32 and stores the result in xmm1. |
| VEX.NDS.128.F3.0F 53 /r VRCPS xmm1, xmm2, xmm3/m32 | V/V | AVX | Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the result in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. |

Description

Computes of an approximate reciprocal 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 operand and the destination operand are XMM registers. The three high-order doublewords of the destination operand are copied from the same bits of the first source operand. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| < 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

Tiny results are always flushed to 0.0, with the sign of the operand:

- The result is guaranteed not to be tiny for inputs that are not greater than $(2^{125}) * (2 - 3 * 2^{-10})$ in absolute value.
- The result is guaranteed to be flushed to 0 for values greater than $(2^{126}) * (1 + 3 * 2^{-11})$ in absolute value.
- Input values in between this range may or may not produce tiny results, depending on the implementation.

INSTRUCTION SET REFERENCE

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

Operation

VRCPS (VEX.128 encoded version)

$DEST[31:0] \leftarrow APPROXIMATE(1/SRC2[31:0])$

$DEST[127:32] \leftarrow SRC1[127:32]$

$DEST[255:128] \leftarrow 0$

RCPSS (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow APPROXIMATE(1/SRC[31:0])$

$DEST[255:32]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS `__m128 _mm_rcp_ss(__m128 a)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

RSQRTPS - Compute Approximate Reciprocals of Square Roots of Packed Single-Precision Floating-point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 0F 52 /r RSQRTPS xmm1, xmm2/m128 | V/V | SSE | Computes the approximate reciprocals of the square roots of packed single-precision values in xmm2/mem and stores the results in xmm1 |
| VEX.128.0F 52 /r VRSQRTPS xmm1, xmm2/m128 | V/V | AVX | Computes the approximate reciprocals of the square roots of packed single-precision values in xmm2/mem and stores the results in xmm1 |
| VEX.256.0F 52 /r VRSQRTPS ymm1, ymm2/m256 | V/V | AVX | Computes the approximate reciprocals of the square roots of packed single-precision values in ymm2/mem and stores the results in ymm1 |

Description

Performs an SIMD computation of the approximate reciprocals of the square roots of the four or eight packed single precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1 for an illustration of an SIMD single-precision floating-point operation.

$$|\text{Relative Error}| < 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register.

When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

When a source value is a negative value (other than 0.0), a floating-point indefinite is returned.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

VRSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC2[255:224]))
```

VRSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[255:128] ← 0
```

RSQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
RSQRTPS __m256 _mm256_rsqrtps (__m256 a);
```

RSQRTPS __m128 _mm_rsqrtps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

RSQRTSS - Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F3 0F 52 /r RSQRTSS xmm1, xmm2/m32 | V/V | SSE | Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1. |
| VEX.NDS.128.F3.0F 52 /r VRSQRTSS xmm1, xmm2, xmm3/m32 | V/V | AVX | Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. |

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the second source operand stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination operand are copied from the same bits of the first source operand. See *Figure 10-6* in the *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating point operation. The relative error for this approximation is:

$$\text{Relative Error} | < 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register.

When a source value is a 0.0, an Inf of the sign of the source value is returned.

A denormal source value is treated as a 0.0 (of the same sign).

When a source value is a negative value (other than 0.0), a floating-point indefinite is returned.

When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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

Operation

VRSQRTSS (VEX.128 encoded version)

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

DEST[127:32] ← SRC1[31:0]

DEST[255:128] ← 0

RSQRTSS (128-bit Legacy SSE version)

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

DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

ROUNDPD- Round Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 09 /r ib ROUNDPD xmm1, xmm2/m128, imm8 | V/V | SSE4_1 | Round packed double-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8 |
| VEX.128.66.0F3A 09 /r ib VROUNDPD xmm1, xmm2/m128, imm8 | V/V | AVX | Round packed double-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8 |
| VEX.256.66.0F3A 09 /r ib VROUNDPD ymm1, ymm2/m256, imm8 | V/V | AVX | Round packed double-precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8 |

Description

Round the four double-precision floating-point values in the source operand (second operand) by the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds the input to an integral value and returns the result as a double-precision floating-point value. 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.

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

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 (255:128) of the corresponding YMM 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 (255:128) of the corresponding YMM register destination are unmodified.

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

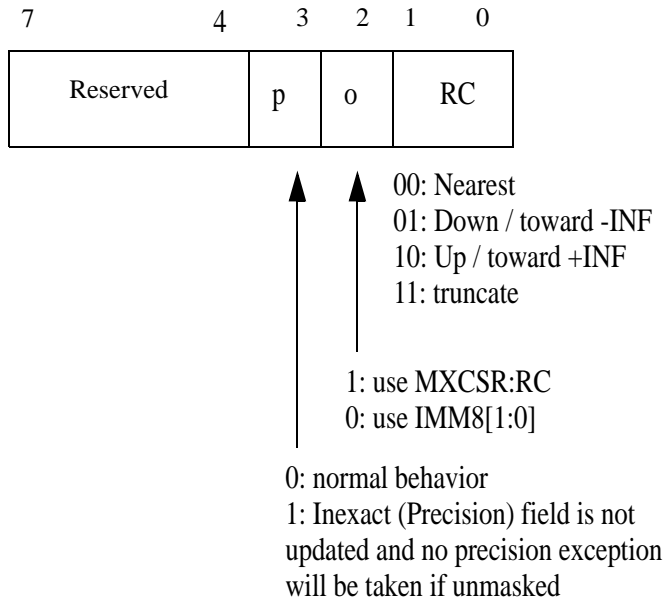


Figure 5-32. VROUNDPD immediate control field definition

Operation

```

RoundToNearestIntegralValue(value, control) {
    rounding_direction ← MXCSR:RC
    if (control[2] == 1)
        rounding_direction ← MXCSR:RC
    else
        rounding_direction ← control[1:0]
    fi
    case (rounding_direction)
        00: dest ← round_to_nearest_even_integer(value)
        01: dest ← round_to_equal_or_smaller_integer(value)
    endcase
}
  
```

INSTRUCTION SET REFERENCE

```
10: dest ← round_to_equal_or_larger_integer(value)
11: dest ← round_to_nearest_smallest_magnitude_integer(value)
esac
```

```
if (control[3] = 0)
{
    if (value != dest)
    {
        set_precision()
    }
}
return(dest)
}
```

VROUNDPD (VEX.256 encoded version)

```
DEST[63:0] ← RoundToNearestIntegralValue(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToNearestIntegralValue(SRC[127:64], ROUND_CONTROL)
DEST[191:128] ← RoundToNearestIntegralValue(SRC[191:128], ROUND_CONTROL)
DEST[255:192] ← RoundToNearestIntegralValue(SRC[255:192], ROUND_CONTROL)
```

VROUNDPD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToNearestIntegralValue(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToNearestIntegralValue(SRC[127:64], ROUND_CONTROL)
DEST[255:128] ← 0
```

ROUNDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToNearestIntegralValue(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToNearestIntegralValue(SRC[127:64], ROUND_CONTROL)
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
__m256 _mm256_round_pd(__m256d s1, int iRoundMode);
```

```
__m256 _mm256_floor_pd(__m256d s1);
```

```
__m256 _mm256_ceil_pd(__m256d s1)
```

```
__m128 _mm_round_pd(__m128d s1, int iRoundMode);
```

```
__m128 _mm_floor_pd(__m128d s1);
```

```
__m128 _mm_ceil_pd(__m128d s1)
```

SIMD Floating-Point Exceptions

Precision, Invalid

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

ROUNDPS- Round Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 08 /r ib ROUNDPS xmm1, xmm2/m128, imm8 | V/V | SSE4_1 | Round packed single-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8 |
| VEX.128.66.0F3A 08 /r ib VROUNDPS xmm1, xmm2/m128, imm8 | V/V | AVX | Round packed single-precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8 |
| VEX.256.66.0F3A 08 /r ib VROUNDPS ymm1, ymm2/m256, imm8 | V/V | AVX | Round packed single-precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8 |

Description

Round the four or eight single-precision floating-point values in the source operand (second operand) by the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds the input to an integral value and returns the result as a single-precision floating-point value. 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.

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

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 (255:128) of the corresponding YMM 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 (255:128) of the corresponding YMM register destination are unmodified.

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

Operation

(see ROUNDPPD for definition of RoundToNearestIntegralValue)

VROUNDPS (VEX.256 encoded version)

DEST[31:0] ← RoundToNearestIntegralValue(SRC[31:0], ROUND_CONTROL)
 DEST[63:32] ← RoundToNearestIntegralValue(SRC[63:32], ROUND_CONTROL)
 DEST[95:64] ← RoundToNearestIntegralValue(SRC[95:64]], ROUND_CONTROL)
 DEST[127:96] ← RoundToNearestIntegralValue(SRC[127:96]], ROUND_CONTROL)
 DEST[159:128] ← RoundToNearestIntegralValue(SRC[159:128]], ROUND_CONTROL)
 DEST[191:160] ← RoundToNearestIntegralValue(SRC[191:160]], ROUND_CONTROL)
 DEST[223:192] ← RoundToNearestIntegralValue(SRC[223:192]], ROUND_CONTROL)
 DEST[255:224] ← RoundToNearestIntegralValue(SRC[255:224]], ROUND_CONTROL)

VROUNDPS (VEX.128 encoded version)

DEST[31:0] ← RoundToNearestIntegralValue(SRC[31:0], ROUND_CONTROL)
 DEST[63:32] ← RoundToNearestIntegralValue(SRC[63:32], ROUND_CONTROL)
 DEST[95:64] ← RoundToNearestIntegralValue(SRC[95:64]], ROUND_CONTROL)
 DEST[127:96] ← RoundToNearestIntegralValue(SRC[127:96]], ROUND_CONTROL)
 DEST[255:128] ← 0

ROUNDPS(128-bit Legacy SSE version)

DEST[31:0] ← RoundToNearestIntegralValue(SRC[31:0], ROUND_CONTROL)
 DEST[63:32] ← RoundToNearestIntegralValue(SRC[63:32], ROUND_CONTROL)
 DEST[95:64] ← RoundToNearestIntegralValue(SRC[95:64]], ROUND_CONTROL)
 DEST[127:96] ← RoundToNearestIntegralValue(SRC[127:96]], ROUND_CONTROL)
 DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

```
__m256 _mm256_round_ps(__m256 s1, int iRoundMode);
```

```
__m256 _mm256_floor_ps(__m256 s1);
```

```
__m256 _mm256_ceil_ps(__m256 s1)
```

```
__m128 _mm_round_ps(__m128 s1, int iRoundMode);
```

```
__m128 _mm_floor_ps(__m128 s1);
```

```
__m128 _mm_ceil_ps(__m128 s1)
```

INSTRUCTION SET REFERENCE

SIMD Floating-Point Exceptions

Precision, Invalid

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

ROUNDSD - Round Scalar Double-Precision Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 3A 0B /r ib ROUNDSD xmm1, xmm2/m64, imm8 | V/V | SSE4_1 | Round the low packed double precision floating-point value in xmm2/m64 and place the result in xmm1. The rounding mode is determined by imm8. |
| VEX.NDS.128.66.0F3A 0B /r ib VROUNDSD xmm1, xmm2, xmm3/m64, imm8 | V/V | AVX | Round the low packed double precision floating-point value in xmm3/m64 and place the result in xmm1. The rounding mode is determined by imm8. Upper packed double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64]. |

Description

Round the DP FP value in the second source operand by the rounding mode specified in the immediate operand and place the result in the destination operand. The rounding process rounds the lowest double precision floating-point input to an integral value and returns the result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained. The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 5-32. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit. 2 selects the source of rounding mode control. Bits 1:0 specify a nonsticky rounding-mode value (Figure 5-32 lists the encoded values for rounding-mode field). The precision floating-point exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed. Software should ensure VROUNDSD is encoded with VEX.L=0. Encoding VROUNDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VROUNDSD (VEX.128 encoded version)

DEST[63:0] ← RoundToNearestIntegralValue(SRC2[63:0], ROUND_CONTROL)

DEST[127:64] ← SRC1[127:64]

DEST[255:128] ← 0

ROUNDSD (128-bit Legacy SSE version)

DEST[63:0] ← RoundToNearestIntegralValue(SRC[63:0], ROUND_CONTROL)

DEST[255:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSD __m128d _mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);

__m128d _mm_floor_sd(__m128d dst, __m128d s1);

__m128d _mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN), Precision (signaled only if imm[3] == '0'; if imm[3] == '1', then the Precision Mask in the MXCSR is ignored.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Exceptions Type 3

ROUNDSS - Round Scalar Single-Precision Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 66 0F 3A 0A ib ROUNDSS xmm1, xmm2/m32, imm8 | V/V | SSE4_1 | Round the low packed single precision floating-point value in xmm2/m32 and place the result in xmm1. The rounding mode is determined by imm8. |
| VEX.NDS.128.66.0F3A 0A ib VROUNDSS xmm1, xmm2, xmm3/m32, imm8 | V/V | AVX | Round the low packed single precision floating-point value in xmm3/m32 and place the result in xmm1. The rounding mode is determined by imm8. Also, upper packed single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. |

Description

Round the single precision floating-point value in the second source operand by the rounding mode specified in the immediate operand and place the result in the destination operand. The rounding process rounds the lowest single precision floating-point input to an integral value and returns the result as a single precision floating-point value in the lowest position. The upper three single precision floating-point values in the destination are copied from the first source operand. The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 5-32. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit. 2 selects the source of rounding mode control.

Bits 1:0 specify a non-sticky rounding-mode value (Table 5-1 lists the encoded values for rounding-mode field). The precision floating-point exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255: 32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255: 128) of the destination YMM register are zeroed.

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

Operation

VROUNDSS (VEX.128 encoded version)

DEST[31:0] \leftarrow RoundToNearestIntegralValue(SRC2[31:0], ROUND_CONTROL)

DEST[127:32] \leftarrow SRC1[127:32]

DEST[255:128] \leftarrow 0

ROUNDSS (128-bit Legacy SSE version)

DEST[31:0] \leftarrow RoundToNearestIntegralValue(SRC[31:0], ROUND_CONTROL)

DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSS __m128 _mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);

__m128 _mm_floor_ss(__m128 dst, __m128 s1);

__m128 _mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN), Precision (signaled only if imm[3] == '0'; if imm[3] == '1', then the Precision Mask in the MXCSR is ignored.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Exceptions Type 3

SHUFPD - Shuffle Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8 | V/V | SSE2 | Shuffle Packed double-precision floating-point values selected by imm8 from xmm1 and xmm2/mem |
| VEX.NDS.128.66.0F C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8 | V/V | AVX | Shuffle Packed double-precision floating-point values selected by imm8 from xmm1 and xmm2/mem |
| VEX.NDS.256.66.0F C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8 | V/V | AVX | Shuffle Packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/mem |

Description

Moves either of the two packed double-precision floating-point values from each double quadword in the first source operand (second operand) into the low quadword of each double quadword of the destination operand (first operand); moves either of the two packed double-precision floating-point values from the second source operand (third operand) into the high quadword of each double quadword of the destination operand (see Figure 5-33). The immediate determines which values are moved to the destination operand.

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

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

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 (255:128) of the corresponding YMM register destination are unmodified.

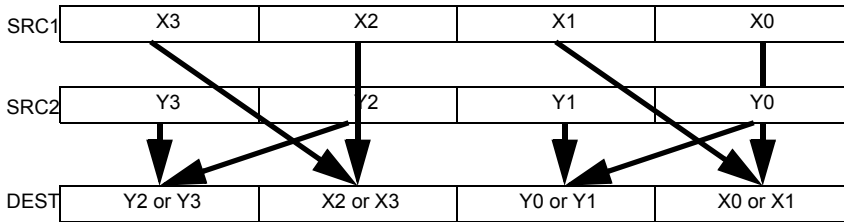


Figure 5-33. VSHUFPD Operation

Operation**VSHUFPD (VEX.256 encoded version)**

```

IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] ← SRC2[191:128]
    ELSE DEST[255:192] ← SRC2[255:192] FI;

```

VSHUFPD (VEX.128 encoded version)

```

IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[255:128] ← 0

```

VSHUFPD (128-bit Legacy SSE version)

```

IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]

```

```
ELSE DEST[63:0] ← SRC1[127:64] FI;  
IF IMM0[1] = 0  
  THEN DEST[127:64] ← SRC2[63:0]  
  ELSE DEST[127:64] ← SRC2[127:64] FI;  
DEST[255:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);  
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

SHUFPS - Shuffle Packed Single Precision Floating-Point Values

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

Description

Moves two of the four packed single-precision floating-point values from each double qword of the first source operand (second operand) into the low quadword of each double qword of the destination operand (first operand); moves two of the four packed single-precision floating-point values from each double qword of the second source operand (third operand) into to the high quadword of each double qword of the destination operand (see Figure 5-34). The selector operand (third operand) determines which values are moved to the destination operand.

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

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

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 (255:128) of the corresponding YMM register destination are unmodified.

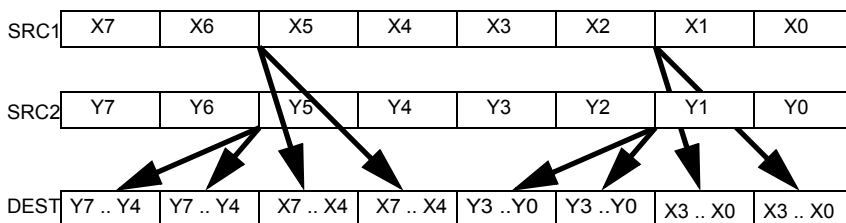


Figure 5-34. VSHUFPS Operation

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
}

```

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]);

```

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[255:128] ← 0

```


INSTRUCTION SET REFERENCE

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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

SQRTPD- Square Root of Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 51 /r SQRTPD xmm1, xmm2/m128 | 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 51 /r VSQRTPD xmm1, xmm2/m128 | 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 51/r VSQRTPD ymm1, ymm2/m256 | V/V | AVX | Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1 |

Description

Performs an SIMD computation of the square roots of the two or four packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand.

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

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 (255: 128) of the corresponding YMM 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 (255: 128) of the corresponding YMM register destination are unmodified.

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

Operation

VSQRTPD (VEX.256 encoded version)

INSTRUCTION SET REFERENCE

DEST[63:0] ← SQR(TSRC[63:0])
DEST[127:64] ← SQR(TSRC[127:64])
DEST[191:128] ← SQR(TSRC[191:128])
DEST[255:192] ← SQR(TSRC[255:192])

.

VSQRTPD (VEX.128 encoded version)

DEST[63:0] ← SQR(TSRC[63:0])
DEST[127:64] ← SQR(TSRC[127:64])
DEST[255:128] ← 0

SQRTPD (128-bit Legacy SSE version)

DEST[63:0] ← SQR(TSRC[63:0])
DEST[127:64] ← SQR(TSRC[127:64])
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPD __m256d _mm256_sqrt_pd (__m256d a);

SQRTPD __m128d _mm_sqrt_pd (__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

SQRTPS- Square Root of Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 51 /r SQRTPS xmm1, xmm2/m128 | V/V | SSE | Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1 |
| VEX.128.0F 51 /r VSQRTPS xmm1, xmm2/m128 | V/V | AVX | Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1 |
| VEX.256.0F 51/r VSQRTPS ymm1, ymm2/m256 | V/V | AVX | Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1 |

Description

Performs an SIMD computation of the square roots of the four or eight packed single-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand.

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

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 (255: 128) of the corresponding YMM 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 (255: 128) of the corresponding YMM register destination are unmodified.

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

Operation

VSQRTPS (VEX.256 encoded version)

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

INSTRUCTION SET REFERENCE

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPS __m256 _mm256_sqrt_ps (__m256 a);

SQRTPS __m128 _mm_sqrt_ps (__m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

SQRTSD - Compute Square Root of Scalar Double-Precision Floating-Point Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F2 0F 51/ SQRTSD xmm1,xmm2/m64 | 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 51/ VSQRTSD xmm1,xmm2, xmm3/m64 | V/V | AVX | Computes square root of the low double-precision floating point value in xmm3/m64 and stores the results in xmm2. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64]. |

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. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar double-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

DEST[63:0] ← SQRT(SRC2[63:0])

DEST[127:64] ← SRC1[127:64]

DEST[255:128] ← 0

SQRTSD (128-bit Legacy SSE version)

DEST[63:0] ← SQRT(SRC[63:0])

INSTRUCTION SET REFERENCE

DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

SQRTSS - Compute Square Root of Scalar Single-Precision Value

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| F3 0F 51 SQRTSS xmm1, xmm2/m32 | 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 51 VSQRTSS xmm1, xmm2, xmm3/m32 | 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]. |

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. The three high order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a scalar single-precision floating-point operation.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

DEST[31:0] ← SQRT(SRC2[31:0])

DEST[127:32] ← SRC1[127:32]

DEST[255:128] ← 0

SQRTSS (128-bit Legacy SSE version)

INSTRUCTION SET REFERENCE

DEST[31:0] ← SQR2(SRC2[31:0])

DEST[255:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

SQR2SS __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VSTMXCSR—Store MXCSR Register State

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|-----------------------|--|
| VEX.128.0F AE /3 VSTMXCSR <i>m32</i> | V/V | AVX | Store contents of MXCSR register to <i>m32</i> . |

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

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

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

Operation

$m32 \leftarrow \text{MXCSR};$

Intel C/C++ Compiler Intrinsic Equivalent

`_mm_getcsr(void)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 9

SUBPD- Subtract Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 5C /r SUBPD xmm1, xmm2/m128 | V/V | SSE2 | Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and stores result in xmm1 |
| VEX.NDS.128.66.0F 5C /r VSUBPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1 |
| VEX.NDS.256.66.0F 5C /r VSUBPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1 |

Description

Performs an SIMD subtract of the 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.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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VSUBPD (VEX.256 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] - \text{SRC2}[127:64]$$

$$\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] - \text{SRC2}[191:128]$$

$$\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] - \text{SRC2}[255:192]$$

VSUBPD (VEX.128 encoded version)

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

SUBPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] - \text{SRC}[127:64]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPD `__m256d _mm256_sub_pd (__m256d a, __m256d b);`

SUBPD `__m128d _mm_sub_pd (__m128d a, __m128d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

SUBPS- Subtract Packed Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 0F 5C /r SUBPS xmm1, xmm2/m128 | V/V | SSE | Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and stores result in xmm1 |
| VEX.NDS.128.0F 5C /r VSUBPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1 |
| VEX.NDS.256.0F 5C /r VSUBPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1 |

Description

Performs an SIMD subtract of the eight or sixteen 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.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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

VSUBPS (VEX.256 encoded version)

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$$

$$\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$$

$$\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$$

$$\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$$

$\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] - \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] - \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] - \text{SRC2}[255:224]$.

VSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

SUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPS `__m256 _mm256_sub_ps (__m256 a, __m256 b);`

SUBPS `__m128 _mm_sub_ps (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

SUBSD- Subtract Scalar Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F2 0F 5C /r SUBSD xmm1, xmm2/m64 | V/V | SSE2 | Subtract the low double-precision floating-point value in xmm2/mem from xmm1 and store the result in xmm1 |
| VEX.NDS.128.F2.0F 5C /r VSUBSD xmm1,xmm2, xmm3/m64 | V/V | AVX | Subtract the low double-precision floating-point value in xmm3/mem from xmm2 and store the result in xmm1 |

Description

Subtract the low double-precision floating-point values in the second source operand from 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 destination and first source operand are the same. Bits (255:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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 (VEX.128 encoded version)

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$$

$$\text{DEST}[255:128] \leftarrow 0$$
SUBSD (128-bit Legacy SSE version)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$$

$$\text{DEST}[255:64] \text{ (Unmodified)}$$

Intel C/C++ Compiler Intrinsic Equivalent

SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

SUBSS- Subtract Scalar Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| F3 0F 5C /r SUBSS xmm1, xmm2/m32 | V/V | SSE | Subtract the low single-precision floating-point value in xmm2/mem from xmm1 and store the result in xmm1 |
| VEX.NDS.128.F3.0F 5C /r VSUBSS xmm1,xmm2, xmm3/m32 | V/V | AVX | Subtract the low single-precision floating-point value in xmm3/mem from xmm2 and store the result in xmm1 |

Description

Subtract the low single-precision floating-point values from the second source operand and the first source operand and store the double-precision floating-point result in the destination operand.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (255:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (255:128) of the destination YMM register are zeroed.

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

VSUBSS (VEX.128 encoded version)

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$$

$$\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$$

$$\text{DEST}[255:128] \leftarrow 0$$
SUBSS (128-bit Legacy SSE version)

$$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] - \text{SRC}[31:0]$$

$$\text{DEST}[255:32] \text{ (Unmodified)}$$

Intel C/C++ Compiler Intrinsic Equivalent

SUBSS `__m128 _mm_sub_ss (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 2E /r UCOMISD xmm1, xmm2/m64 | 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 2E /r VUCOMISD xmm1, xmm2/m64 | V/V | AVX | Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly. |

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 (#1) 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: In VEX-encoded versions, VEX.vvvv is 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

UCOMISD (all versions)

RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) {

(* Set EFLAGS *) CASE (RESULT) OF

UNORDERED: ZF,PF,CF \leftarrow 111;

GREATER_THAN: ZF,PF,CF \leftarrow 000;

LESS_THAN: ZF,PF,CF \leftarrow 001;

EQUAL: ZF,PF,CF \leftarrow 100;

ESAC;

OF, AF, SF \leftarrow 0; }

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_ucomieq_sd(__m128d a, __m128d b)

int _mm_ucomilt_sd(__m128d a, __m128d b)

int _mm_ucomile_sd(__m128d a, __m128d b)

int _mm_ucomigt_sd(__m128d a, __m128d b)

int _mm_ucomige_sd(__m128d a, __m128d b)

int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv \neq 1111B.

UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 2E /r UCOMISS xmm1, xmm2/m32 | V/V | SSE | Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |
| VEX.128.0F 2E /r VUCOMISS xmm1, xmm2/m32 | V/V | AVX | Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |

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 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: In VEX-encoded versions, VEX.vvvv is 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

UCOMISS (all versions)

```
RESULT ← UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
```

(* Set EFLAGS *) CASE (RESULT) OF

UNORDERED: ZF,PF,CF \leftarrow 111;

GREATER_THAN: ZF,PF,CF \leftarrow 000;

LESS_THAN: ZF,PF,CF \leftarrow 001;

EQUAL: ZF,PF,CF \leftarrow 100;

ESAC;

OF, AF, SF \leftarrow 0; }

Intel C/C++ Compiler Intrinsic Equivalent

int __mm_ucomieq_ss(__m128 a, __m128 b)

int __mm_ucomilt_ss(__m128 a, __m128 b)

int __mm_ucomile_ss(__m128 a, __m128 b)

int __mm_ucomigt_ss(__m128 a, __m128 b)

int __mm_ucomige_ss(__m128 a, __m128 b)

int __mm_ucomineq_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

UNPCKHPD- Unpack and Interleave High Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|---|
| 66 0F 15 /r UNPCKHPD xmm1, xmm2/m128 | V/V | SSE2 | Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128. |
| VEX.NDS.128.66.0F 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128. |
| VEX.NDS.256.66.0F 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256 | V/V | AVX | Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256. |

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 versions

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

VUNPCKHPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[127:64]
 DEST[127:64] ← SRC2[127:64]
 DEST[255:128] ← 0

UNPCKHPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[127:64]
 DEST[127:64] ← SRC2[127:64]
 DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)

UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

UNPCKHPS- Unpack and Interleave High Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|---|
| 0F 15 /r UNPCKHPS xmm1, xmm2/m128 | V/V | SSE | Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128. |
| VEX.NDS.128.0F 15 /r VUNPCKHPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128. |
| VEX.NDS.256.0F 15 /r VUNPCKHPS ymm1,ymm2,ymm3/m256 | V/V | AVX | Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256. |

Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

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

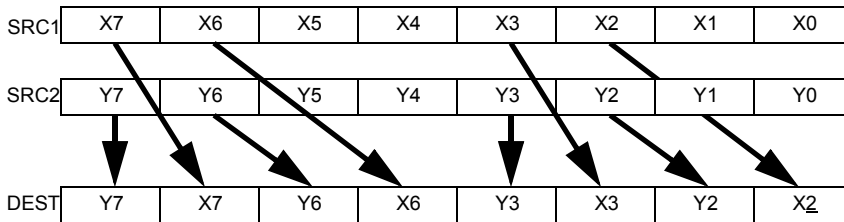


Figure 5-35. VUNPCKHPS Operation

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VUNPCKHPS (VEX.256 encoded version)

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

VUNPCKHPS (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[95:64]$
 $DEST[63:32] \leftarrow SRC2[95:64]$
 $DEST[95:64] \leftarrow SRC1[127:96]$
 $DEST[127:96] \leftarrow SRC2[127:96]$
 $DEST[255:128] \leftarrow 0$

UNPCKHPS (128-bit Legacy SSE version)

INSTRUCTION SET REFERENCE

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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);

UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

UNPCKLPD- Unpack and Interleave Low Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 14 /r UNPCKLPD xmm1, xmm2/m128 | V/V | SSE2 | Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128. |
| VEX.NDS.128.66.0F 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128 | V/V | AVX | Unpacks and Interleaves double precision floating-point values low high quadwords of xmm2 and xmm3/m128. |
| VEX.NDS.256.66.0F 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256 | V/V | AVX | Unpacks and Interleaves double precision floating-point values low high quadwords of ymm2 and ymm3/m256. |

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 versions:

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

VUNPCKLPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0]

INSTRUCTION SET REFERENCE

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

VUNPCKLPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] ← 0

UNPCKLPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)

UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

UNPCKLPS- Unpack and Interleave Low Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 14 /r UNPCKLPS xmm1, xmm2/m128 | V/V | SSE | Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128. |
| VEX.NDS.128.0F 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128. |
| VEX.NDS.256.0F 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256 | V/V | AVX | Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256. |

Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand

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

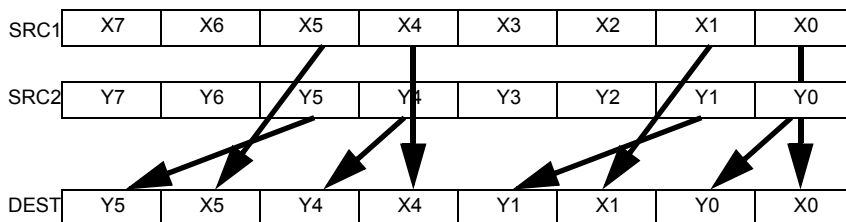


Figure 5-36. VUNPCKLPS Operation

VEX.128 encoded version: the first source operand second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: T second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS `__m256 _mm256_unpacklo_ps (__m256 a, __m256 b);`

UNPCKLPS `__m128 _mm_unpacklo_ps (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|--|------------------------------|--------------------------|--|
| 66 0F 57/r XORPD xmm1, xmm2/m128 | 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 57 /r VXORPD xmm1,xmm2, xmm3/m128 | 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 57 /r VXORPD ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

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[255:128] ← 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[255:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VXORPD `__m256d _mm256_xor_pd (__m256d a, __m256d b);`

XORPD `__m128d _mm_xor_pd (__m128d a, __m128d b);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|------------------------------|--------------------------|--|
| 0F 57 /r XORPS xmm1, xmm2/m128 | V/V | SSE | Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem |
| VEX.NDS.128.0F 57 /r VXORPS xmm1,xmm2, xmm3/m128 | V/V | AVX | Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem |
| VEX.NDS.256.0F 57 /r VXORPS ymm1, ymm2, ymm3/m256 | V/V | AVX | Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem |

Description

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

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 second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (255:128) of the corresponding 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 (255:128) of the corresponding YMM register destination are unmodified.

Operation

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]

$\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE XOR } \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE XOR } \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE XOR } \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE XOR } \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE XOR } \text{SRC2}[255:224]$.

VXORPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE XOR } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE XOR } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE XOR } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE XOR } \text{SRC2}[127:96]$
 $\text{DEST}[255:128] \leftarrow 0$

XORPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE XOR } \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE XOR } \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE XOR } \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE XOR } \text{SRC2}[127:96]$
 $\text{DEST}[255:128]$ (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VXORPS `__m256 _mm256_xor_ps (__m256 a, __m256 b);`

XORPS `__m128 _mm_xor_ps (__m128 a, __m128 b);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VZEROALL- Zero All YMM registers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---------------------------|------------------------------|--------------------------|------------------------|
| VEX.256.0F 77 VZEROALL | V/V | AVX | Zero all YMM registers |

Description

The instruction zeros contents of all XMM or YMM registers.

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

In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROALL (VEX.256 encoded version)

IF (64-bit mode)

YMM0[255:0] ← 0

YMM1[255:0] ← 0

YMM2[255:0] ← 0

YMM3[255:0] ← 0

YMM4[255:0] ← 0

YMM5[255:0] ← 0

YMM6[255:0] ← 0

YMM7[255:0] ← 0

YMM8[255:0] ← 0

YMM9[255:0] ← 0

YMM10[255:0] ← 0

YMM11[255:0] ← 0

YMM12[255:0] ← 0

YMM13[255:0] ← 0

YMM14[255:0] ← 0

YMM15[255:0] ← 0

ELSE

YMM0[255:0] ← 0

YMM1[255:0] ← 0

YMM2[255:0] ← 0

YMM3[255:0] ← 0

YMM4[255:0] ← 0

YMM5[255:0] ← 0

YMM6[255:0] ← 0

YMM7[255:0] ← 0

YMM8-15: Unmodified

FI

Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL _mm256_zeroall()

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 8

VZEROUPPER- Zero Upper bits of YMM registers

| Opcode/ Instruction | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|-----------------------------|------------------------------|--------------------------|--|
| VEX.128.0F 77 VZEROUPPER | V/V | AVX | Zero upper 128 bits of all YMM registers |

Description

The instruction zeros the upper 128 bits of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

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

In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROUPPER

IF (64-bit mode)

YMM0[255:128] ← 0
 YMM1[255:128] ← 0
 YMM2[255:128] ← 0
 YMM3[255:128] ← 0
 YMM4[255:128] ← 0
 YMM5[255:128] ← 0
 YMM6[255:128] ← 0
 YMM7[255:128] ← 0
 YMM8[255:128] ← 0
 YMM9[255:128] ← 0
 YMM10[255:128] ← 0
 YMM11[255:128] ← 0
 YMM12[255:128] ← 0
 YMM13[255:128] ← 0
 YMM14[255:128] ← 0
 YMM15[255:128] ← 0

ELSE

YMM0[255:128] ← 0
 YMM1[255:128] ← 0
 YMM2[255:128] ← 0
 YMM3[255:128] ← 0

YMM4[255:128] ← 0
YMM5[255:128] ← 0
YMM6[255:128] ← 0
YMM7[255:128] ← 0
YMM8-15: unmodified

FI

Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER _mm256_zeroupper()

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 8

CHAPTER 6 INSTRUCTION SET REFERENCE - FMA

6.1 FMA INSTRUCTION SET REFERENCE

This section describes FMA instructions in details. Conventions and notations of instruction format can be found in Section 5.1.

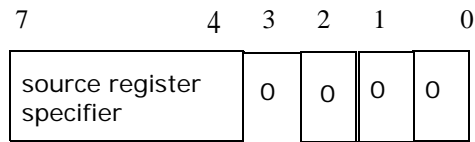
VFMADDPD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 69 /r /is4 VFMADDPD xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 69 /r /is4 VFMADDPD xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 69 /r /is4 VFMADDPD ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 69 /r /is4 VFMADDPD ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm0. |

Description

Performs a SIMD multiply of the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the second source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand. An 8-bit imme-

mediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.



Bits[3:0]: Reserved for Future Use and must be Zero

Figure 6-1. VFMADDPD Immediate Byte

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDPD DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\text{DEST}[63:0] \leftarrow (\text{SRC1}[63:0] * \text{SRC2}[63:0] + \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (\text{SRC1}[127:64] * \text{SRC2}[127:64] + \text{SRC3}[127:64])$$

$$\text{DEST}[191:128] \leftarrow (\text{SRC1}[191:128] * \text{SRC2}[191:128] + \text{SRC3}[191:128])$$

$$\text{DEST}[255:192] \leftarrow (\text{SRC1}[255:192] * \text{SRC2}[255:192] + \text{SRC3}[255:192])$$

VFMADDPD DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\text{DEST}[63:0] \leftarrow (\text{SRC1}[63:0] * \text{SRC2}[63:0] + \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (\text{SRC1}[127:64] * \text{SRC2}[127:64] + \text{SRC3}[127:64])$$

$$\text{DEST}[255:128] \leftarrow 0$$

INSTRUCTION SET REFERENCE - FMA

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c, const int ctrl);

VFMADDPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDPS - Fused Multiply-Add of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 68 /r /is4 VFMADDPS xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 68 /r /is4 VFMADDPS xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 68 /r /is4 VFMADDPS ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 68 /r /is4 VFMADDPS ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm0. |

Description

Performs a SIMD multiply of the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the second source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or 8 packed single-precision floating-point values to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDPS DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] + \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (\text{SRC1}[63:32] * \text{SRC2}[63:32] + \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (\text{SRC1}[95:64] * \text{SRC2}[95:64] + \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (\text{SRC1}[127:96] * \text{SRC2}[127:96] + \text{SRC3}[127:96]) \\ \text{DEST}[159:128] &\leftarrow (\text{SRC1}[159:128] * \text{SRC2}[159:128] + \text{SRC3}[159:128]) \\ \text{DEST}[191:160] &\leftarrow (\text{SRC1}[191:160] * \text{SRC2}[191:160] + \text{SRC3}[191:160]) \\ \text{DEST}[223:192] &\leftarrow (\text{SRC1}[223:192] * \text{SRC2}[223:192] + \text{SRC3}[223:192]) \\ \text{DEST}[255:224] &\leftarrow (\text{SRC1}[255:224] * \text{SRC2}[255:224] + \text{SRC3}[255:224]) \end{aligned}$$

VFMADDPS DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] + \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (\text{SRC1}[63:32] * \text{SRC2}[63:32] + \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (\text{SRC1}[95:64] * \text{SRC2}[95:64] + \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (\text{SRC1}[127:96] * \text{SRC2}[127:96] + \text{SRC3}[127:96]) \\ \text{DEST}[255:128] &\leftarrow 0 \end{aligned}$$

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c, const int ctrl);

VFMADDPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDSD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 6B /r /is4 VFMADDSD xmm0, xmm1, xmm2/m64, xmm3 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 6B /r /is4 VFMADDSD xmm0, xmm1, xmm2, xmm3/m64 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |

Description

Performs an SIMD multiply of the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the second 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. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:64] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSD DEST, SRC1, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow (\text{SRC1}[63:0] * \text{SRC2}[63:0] + \text{SRC3}[63:0])$$

$$\text{DEST}[255:64] \leftarrow 0$$

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADDSS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 6A /r /is4 VFMADDSS xmm0, xmm1, xmm2/m32, xmm3 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 6A /r /is4 VFMADDSS xmm0, xmm1, xmm2, xmm3/m32 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |

Description

Performs an SIMD multiply of the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the second 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. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:32] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSS DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] + \text{SRC3}[31:0])$$

$$\text{DEST}[255:32] \leftarrow 0$$

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADDSUBPD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 5D /r /is4 VFMADDSUBPD xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 5D /r /is4 VFMADDSUBPD xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 5D /r /is4 VFMADDSUBPD ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 5D /r /is4 VFMADDSUBPD ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm0. |

Description

Performs an SIMD multiply of the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the second 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. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the

second source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. If `VEX.W` bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in `VEX.vvvv`) are YMM registers, the third source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If `VEX.W` bit is 0, the first source operand (encoded in `imm8[7:4]`) and third source operand (encoded in `VEX.vvvv`) are XMM registers, the second source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. If `VEX.W` bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in `VEX.vvvv`) are XMM registers, the third source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode `imm8[7]` is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSUBPD DEST, SRC1, SRC2, SRC3 (256-bit version)

$DEST[63:0] \leftarrow (SRC1[63:0]*SRC2[63:0] - SRC3[63:0])$

$DEST[127:64] \leftarrow (SRC1[127:64]*SRC2[127:64] + SRC3[127:64])$

$DEST[191:128] \leftarrow (SRC1[191:128]*SRC2[191:128] - SRC3[191:128])$

$DEST[255:192] \leftarrow (SRC1[255:192]*SRC2[255:192] + SRC3[255:192])$

VFMADDSUBPD DEST, SRC1, SRC2, SRC3 (128-bit version)

$DEST[63:0] \leftarrow (SRC1[63:0]*SRC2[63:0] - SRC3[63:0])$

$DEST[127:64] \leftarrow (SRC1[127:64]*SRC2[127:64] + SRC3[127:64])$

$DEST[255:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

`VFMADDSUBPD __m128d __mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c, const int ctrl);`

`VFMADDSUBPD __m256d __mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c, const int ctrl);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDSUBPS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 5C /r /is4 VFMADDSUBPS xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 5C /r /is4 VFMADDSUBPS xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 5C /r /is4 VFMADDSUBPS ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 5C /r /is4 VFMADDSUBPS ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract ymm3/mem and put result in ymm0. |

Description

Performs an SIMD multiply of the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the second source operand. From the infinite precision intermediate result adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and

second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in `imm8[7:4]`) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode `imm8[7]` is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSUBPS DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] - \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (\text{SRC1}[63:32] * \text{SRC2}[63:32] + \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (\text{SRC1}[95:64] * \text{SRC2}[95:64] - \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (\text{SRC1}[127:96] * \text{SRC2}[127:96] + \text{SRC3}[127:96]) \\ \text{DEST}[159:128] &\leftarrow (\text{SRC1}[159:128] * \text{SRC2}[159:128] - \text{SRC3}[159:128]) \\ \text{DEST}[191:160] &\leftarrow (\text{SRC1}[191:160] * \text{SRC2}[191:160] + \text{SRC3}[191:160]) \\ \text{DEST}[223:192] &\leftarrow (\text{SRC1}[223:192] * \text{SRC2}[223:192] - \text{SRC3}[223:192]) \\ \text{DEST}[255:224] &\leftarrow (\text{SRC1}[255:224] * \text{SRC2}[255:224] + \text{SRC3}[255:224]) \end{aligned}$$

VFMADDSUBPS DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] - \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (\text{SRC1}[63:32] * \text{SRC2}[63:32] + \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (\text{SRC1}[95:64] * \text{SRC2}[95:64] - \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (\text{SRC1}[127:96] * \text{SRC2}[96:64] + \text{SRC3}[127:96]) \\ \text{DEST}[255:128] &\leftarrow 0 \end{aligned}$$

Intel C/C++ Compiler Intrinsic Equivalent

`VFMADDSUBPS __m128 __mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c, const int ctrl);`

`VFMADDSUBPS __m256 __mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c, const int ctrl);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

INSTRUCTION SET REFERENCE - FMA

Other Exceptions

See Exceptions Type 2

VFMSUBADDPD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 5F /r /is4 VFMSUBADDPD xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 5F /r /is4 VFMSUBADDPD xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 5F /r /is4 VFMSUBADDPD ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 5F /r /is4 VFMSUBADDPD ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm0. |

Description

Performs an SIMD multiply of the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the second 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. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the

second source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. If `VEX.W` bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in `VEX.vvvv`) are YMM registers, the third source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If `VEX.W` bit is 0, the first source operand (encoded in `imm8[7:4]`) and third source operand (encoded in `VEX.vvvv`) are XMM registers, the second source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. If `VEX.W` bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in `VEX.vvvv`) are XMM registers, the third source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode `imm8[7]` is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBADDPD `DEST, SRC1, SRC2, SRC3` (256-bit version)

```
DEST[63:0] ← (SRC1[63:0]*SRC2[63:0] + SRC3[63:0])
DEST[127:64] ← (SRC1[127:64]*SRC2[127:64] - SRC3[127:64])
DEST[191:128] ← (SRC1[191:128]*SRC2[191:128] + SRC3[191:128])
DEST[255:192] ← (SRC1[255:192]*SRC2[255:192] - SRC3[255:192])
```

VFMSUBADDPD `DEST, SRC1, SRC2, SRC3` (128-bit version)

```
DEST[63:0] ← (SRC1[63:0]*SRC2[63:0] + SRC3[63:0])
DEST[127:64] ← (SRC1[127:64]*SRC2[127:64] - SRC3[127:64])
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBADDPD __m128d __mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c, const int ctrl);
VFMSUBADDPD __m256d __mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c, const int ctrl);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADDPS - Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 5E /r /is4 VFMSUBADDPS xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 5E /r /is4 VFMSUBADDPS xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 5E /r /is4 VFMSUBADDPS ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 5E /r /is4 VFMSUBADDPS ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add ymm3/mem and put result in ymm0. |

Description

Performs a SIMD multiply of the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the second source operand. From the infinite precision intermediate result subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and

second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBADDPS DEST, SRC1, SRC2, SRC3 (256-bit version)

```
DEST[31:0] ← (SRC1[31:0]*SRC2[31:0] + SRC3[31:0])
DEST[63:32] ← (SRC1[63:32]*SRC2[63:32] - SRC3[63:32])
DEST[95:64] ← (SRC1[95:64]*SRC2[95:64] + SRC3[95:64])
DEST[127:96] ← (SRC1[127:96]*SRC2[127:96] - SRC3[127:96])
DEST[159:128] ← (SRC1[159:128]*SRC2[159:128] + SRC3[159:128])
DEST[191:160] ← (SRC1[191:160]*SRC2[191:160] - SRC3[191:160])
DEST[223:192] ← (SRC1[223:192]*SRC2[223:192] + SRC3[223:192])
DEST[255:224] ← (SRC1[255:224]*SRC2[255:224] - SRC3[255:224])
```

VFMSUBADDPS DEST, SRC1, SRC2, SRC3 (128-bit version)

```
DEST[31:0] ← (SRC1[31:0]*SRC2[31:0] + SRC3[31:0])
DEST[63:32] ← (SRC1[63:32]*SRC2[63:32] - SRC3[63:32])
DEST[95:64] ← (SRC1[95:64]*SRC2[95:64] + SRC3[95:64])
DEST[127:96] ← (SRC1[127:96]*SRC2[127:96] - SRC3[127:96])
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFMSUBADDPS __m128 __mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c, const int ctrl);
```

```
VFMSUBADDPS __m256 __mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c, const int ctrl);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBPD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 6D /r /is4 VFMSUBPD xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 6D /r /is4 VFMSUBPD xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 6D /r /is4 VFMSUBPD ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 6D /r /is4 VFMSUBPD ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm0. |

Description

Performs a SIMD multiply of the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the second source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBPD DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\text{DEST}[63:0] \leftarrow (\text{SRC1}[63:0] * \text{SRC2}[63:0] - \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (\text{SRC1}[127:64] * \text{SRC2}[127:64] - \text{SRC3}[127:64])$$

$$\text{DEST}[191:128] \leftarrow (\text{SRC1}[191:128] * \text{SRC2}[191:128] - \text{SRC3}[191:128])$$

$$\text{DEST}[255:192] \leftarrow (\text{SRC1}[255:192] * \text{SRC2}[255:192] - \text{SRC3}[255:192])$$

VFMSUBPD DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\text{DEST}[63:0] \leftarrow (\text{SRC1}[63:0] * \text{SRC2}[63:0] - \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (\text{SRC1}[127:64] * \text{SRC2}[127:64] - \text{SRC3}[127:64])$$

$$\text{DEST}[255:128] \leftarrow 0$$

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBPD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c, const int ctrl);

VFMSUBPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBPS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 6C /r /is4 VFMSUBPS xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 6C /r /is4 VFMSUBPS xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 6C /r /is4 VFMSUBPS ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm3 and put result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 6C /r /is4 VFMSUBPS ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm0. |

Description

Performs a SIMD multiply of the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the second source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or 8 packed single-precision floating-point values to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are YMM registers, the third source

operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in `imm8[7:4]`) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode `imm8[7]` is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBPS DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] - \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (\text{SRC1}[63:32] * \text{SRC2}[63:32] - \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (\text{SRC1}[95:64] * \text{SRC2}[95:64] - \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (\text{SRC1}[127:96] * \text{SRC2}[127:96] - \text{SRC3}[127:96]) \\ \text{DEST}[159:128] &\leftarrow (\text{SRC1}[159:128] * \text{SRC2}[159:128] - \text{SRC3}[159:128]) \\ \text{DEST}[191:160] &\leftarrow (\text{SRC1}[191:160] * \text{SRC2}[191:160] - \text{SRC3}[191:160]) \\ \text{DEST}[223:192] &\leftarrow (\text{SRC1}[223:192] * \text{SRC2}[223:192] - \text{SRC3}[223:192]) \\ \text{DEST}[255:224] &\leftarrow (\text{SRC1}[255:224] * \text{SRC2}[255:224] - \text{SRC3}[255:224]) \end{aligned}$$

VFMSUBPS DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] - \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (\text{SRC1}[63:32] * \text{SRC2}[63:32] - \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (\text{SRC1}[95:64] * \text{SRC2}[95:64] - \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (\text{SRC1}[127:96] * \text{SRC2}[127:96] - \text{SRC3}[127:96]) \\ \text{DEST}[255:128] &\leftarrow 0 \end{aligned}$$

Intel C/C++ Compiler Intrinsic Equivalent

`VFMSUBPS __m128 __mm_fmsub_ps (__m128 a, __m128 b, __m128 c, const int ctrl);`

`VFMSUBPS __m256 __mm256_fmsub_ps (__m256 a, __m256 b, __m256 c, const int ctrl);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

INSTRUCTION SET REFERENCE - FMA

Other Exceptions

See Exceptions Type 2

VFMSUBSD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 6F /r /is4 VFMSUBSD xmm0, xmm1, xmm2/m64, xmm3 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 6F /r /is4 VFMSUBSD xmm0, xmm1, xmm2, xmm3/m64 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |

Description

Performs an SIMD multiply of the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts 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. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:64] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBSD DEST, SRC1, SRC2, SRC3

$DEST[63:0] \leftarrow (SRC1[63:0] * SRC2[63:0] - SRC3[63:0])$

$DEST[255:64] \leftarrow 0$

INSTRUCTION SET REFERENCE - FMA

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMSUBSS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 6E /r /is4 VFMSUBSS xmm0, xmm1, xmm2/m32, xmm3 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 6E /r /is4 VFMSUBSS xmm0, xmm1, xmm2, xmm3/m32 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |

Description

Performs an SIMD multiply of the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the second source operand. From the infinite precision intermediate result, subtracts 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. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:32] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMSUBSS DEST, SRC1, SRC2, SRC3

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[31:0] * \text{SRC2}[31:0] - \text{SRC3}[31:0])$$

$$\text{DEST}[255:32] \leftarrow 0$$

INSTRUCTION SET REFERENCE - FMA

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADDPD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 79 /r /is4 VFMADDPD xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 79 /r /is4 VFMADDPD xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 79 /r /is4 VFMADDPD ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm3. Put the result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 79 /r /is4 VFMADDPD ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem. Put the result in ymm0. |

Description

Performs a SIMD multiply of the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the second source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting double-precision elements to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the

second source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. If `VEX.W` bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in `VEX.vvvv`) are YMM registers, the third source operand (encoded in `rm_field`) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If `VEX.W` bit is 0, the first source operand (encoded in `imm8[7:4]`) and third source operand (encoded in `VEX.vvvv`) are XMM registers, the second source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. If `VEX.W` bit is 1, the first source operand (encoded in `imm8[7:4]`) and second source operand (encoded in `VEX.vvvv`) are XMM registers, the third source operand (encoded in `rm_field`) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode `imm8[7]` is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMADDPD DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\text{DEST}[63:0] \leftarrow (- (\text{SRC1}[63:0] * \text{SRC2}[63:0]) + \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (- (\text{SRC1}[127:64] * \text{SRC2}[127:64]) + \text{SRC3}[127:64])$$

$$\text{DEST}[191:128] \leftarrow (- (\text{SRC1}[191:128] * \text{SRC2}[191:128]) + \text{SRC3}[191:128])$$

$$\text{DEST}[255:192] \leftarrow (- (\text{SRC1}[255:192] * \text{SRC2}[255:192]) + \text{SRC3}[255:192])$$

VFNMADDPD DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\text{DEST}[63:0] \leftarrow (- (\text{SRC1}[63:0] * \text{SRC2}[63:0]) + \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (- (\text{SRC1}[127:64] * \text{SRC2}[127:64]) + \text{SRC3}[127:64])$$

$$\text{DEST}[255:128] \leftarrow 0$$

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMADDPD __m128d __mm_fmadd_pd (__m128d a, __m128d b, __m128d c, const int ctrl);
```

```
VFNMADDPD __m256d __mm256_fmadd_pd (__m256d a, __m256d b, __m256d c, const int ctrl);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDPS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 78 /r /is4 VFMADDPS xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 78 /r /is4 VFMADDPS xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 78 /r /is4 VFMADDPS ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm3. Put the result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 78 /r /is4 VFMADDPS ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem. Put the result in ymm0. |

Description

Performs a SIMD multiply of the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the second source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting single-precision elements to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and

second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDPS DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (- (\text{SRC1}[31:0]*\text{SRC2}[31:0]) + \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (- (\text{SRC1}[63:32]*\text{SRC2}[63:32]) + \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (- (\text{SRC1}[95:64]*\text{SRC2}[95:64]) + \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (- (\text{SRC1}[127:96]*\text{SRC2}[127:96]) + \text{SRC3}[127:96]) \\ \text{DEST}[159:128] &\leftarrow (- (\text{SRC1}[159:128]*\text{SRC2}[159:128]) + \text{SRC3}[159:128]) \\ \text{DEST}[191:160] &\leftarrow (- (\text{SRC1}[191:160]*\text{SRC2}[191:160]) + \text{SRC3}[191:160]) \\ \text{DEST}[223:192] &\leftarrow (- (\text{SRC1}[223:192]*\text{SRC2}[223:192]) + \text{SRC3}[223:192]) \\ \text{DEST}[255:224] &\leftarrow (- (\text{SRC1}[255:224]*\text{SRC2}[255:224]) + \text{SRC3}[255:224]) \end{aligned}$$

VFMADDPS DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (- (\text{SRC1}[31:0]*\text{SRC2}[31:0]) + \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (- (\text{SRC1}[63:32]*\text{SRC2}[63:32]) + \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (- (\text{SRC1}[95:64]*\text{SRC2}[95:64]) + \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (- (\text{SRC1}[127:96]*\text{SRC2}[127:96]) + \text{SRC3}[127:96]) \\ \text{DEST}[255:128] &\leftarrow 0 \end{aligned}$$

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c, const int ctrl);

VFMADDPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDSD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 7B /r /is4 VFMADDSD xmm0, xmm1, xmm2/m64, xmm3 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 7B /r /is4 VFMADDSD xmm0, xmm1, xmm2, xmm3/m64 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |

Description

Performs an SIMD multiply of the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the second source operand, adds the negated 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 value to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:64] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFMADDSD DEST, SRC1, SRC2, SRC3

$$\text{DEST}[63:0] \leftarrow (- (\text{SRC1}[63:0] * \text{SRC2}[63:0]) + \text{SRC3}[63:0])$$

DEST[255:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADDSS - Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|---|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 7A /r /is4 VFNMADDSS xmm0, xmm1, xmm2/m32, xmm3 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 7A /r /is4 VFNMADDSS xmm0, xmm1, xmm2, xmm3/m32 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |

Description

Performs an SIMD multiply of the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the second source operand, adds the negated 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 value to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:32] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMADDSS DEST, SRC1, SRC2, SRC3 (128-bit version)

$\text{DEST}[31:0] \leftarrow (- (\text{SRC1}[31:0] * \text{SRC2}[31:0]) + \text{SRC3}[31:0])$
 $\text{DEST}[255:32] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDSS `__m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c, const int ctrl);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUBPD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 7D /r /is4 VFNMSUBPD xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 7D /r /is4 VFNMSUBPD xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 7D /r /is4 VFNMSUBPD ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm3. Put the result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 7D /r /is4 VFNMSUBPD ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem. Put the result in ymm0. |

Description

Performs a SIMD multiply of the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the second source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting double-precision elements to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMSUBPD DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\text{DEST}[63:0] \leftarrow (- (\text{SRC1}[63:0] * \text{SRC2}[63:0]) - \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (- (\text{SRC1}[127:64] * \text{SRC2}[127:64]) - \text{SRC3}[127:64])$$

$$\text{DEST}[191:128] \leftarrow (- (\text{SRC1}[191:128] * \text{SRC2}[191:128]) - \text{SRC3}[191:128])$$

$$\text{DEST}[255:192] \leftarrow (- (\text{SRC1}[255:192] * \text{SRC2}[255:192]) - \text{SRC3}[255:192])$$

VFNMSUBPD DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\text{DEST}[63:0] \leftarrow (- (\text{SRC1}[63:0] * \text{SRC2}[63:0]) - \text{SRC3}[63:0])$$

$$\text{DEST}[127:64] \leftarrow (- (\text{SRC1}[127:64] * \text{SRC2}[127:64]) - \text{SRC3}[127:64])$$

$$\text{DEST}[255:128] \leftarrow 0$$

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBPD __m128d __mm_fnmsub_pd (__m128d a, __m128d b, __m128d c, const int ctrl);

VFNMSUBPD __m256d __mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

INSTRUCTION SET REFERENCE - FMA

Other Exceptions

See Exceptions Type 2

VFNMSUBPS - Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|--|
| VEX.NDS.128.66.0F3A.W0 7C /r /is4 VFNMSUBPS xmm0, xmm1, xmm2/m128, xmm3 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 7C /r /is4 VFNMSUBPS xmm0, xmm1, xmm2, xmm3/m128 | V/V | FMA | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66.0F3A.W0 7C /r /is4 VFNMSUBPS ymm0, ymm1, ymm2/m256, ymm3 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm3. Put the result in ymm0. |
| VEX.NDS.256.66.0F3A.W1 7C /r /is4 VFNMSUBPS ymm0, ymm1, ymm2, ymm3/m256 | V/V | FMA | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem. Put the result in ymm0. |

Description

Performs a SIMD multiply of the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the second source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting single-precision elements to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.256 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are YMM registers, the second source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are YMM registers, the third source operand (encoded in rm_field) is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper 128 bits of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNSUBPS DEST, SRC1, SRC2, SRC3 (256-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (- (\text{SRC1}[31:0] * \text{SRC2}[31:0]) - \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (- (\text{SRC1}[63:32] * \text{SRC2}[63:32]) - \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (- (\text{SRC1}[95:64] * \text{SRC2}[95:64]) - \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (- (\text{SRC1}[127:96] * \text{SRC2}[127:96]) - \text{SRC3}[127:96]) \\ \text{DEST}[159:128] &\leftarrow (- (\text{SRC1}[159:128] * \text{SRC2}[159:128]) - \text{SRC3}[159:128]) \\ \text{DEST}[191:160] &\leftarrow (- (\text{SRC1}[191:160] * \text{SRC2}[191:160]) - \text{SRC3}[191:160]) \\ \text{DEST}[223:192] &\leftarrow (- (\text{SRC1}[223:192] * \text{SRC2}[223:192]) - \text{SRC3}[223:192]) \\ \text{DEST}[255:224] &\leftarrow (- (\text{SRC1}[255:224] * \text{SRC2}[255:224]) - \text{SRC3}[255:224]) \end{aligned}$$

VFNSUBPS DEST, SRC1, SRC2, SRC3 (128-bit version)

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (- (\text{SRC1}[31:0] * \text{SRC2}[31:0]) - \text{SRC3}[31:0]) \\ \text{DEST}[63:32] &\leftarrow (- (\text{SRC1}[63:32] * \text{SRC2}[63:32]) - \text{SRC3}[63:32]) \\ \text{DEST}[95:64] &\leftarrow (- (\text{SRC1}[95:64] * \text{SRC2}[95:64]) - \text{SRC3}[95:64]) \\ \text{DEST}[127:96] &\leftarrow (- (\text{SRC1}[127:96] * \text{SRC2}[96:64]) - \text{SRC3}[127:96]) \\ \text{DEST}[255:128] &\leftarrow 0 \end{aligned}$$

intel C/C++ Compiler Intrinsic Equivalent

```
VFNSUBPS __m128 __mm_fnmsub_ps (__m128 a, __m128 b, __m128 c, const int ctrl);
```

VFNMSUBPS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSSUBSD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 7F /r /is4 VFNMSSUBSD xmm0, xmm1, xmm2/m64, xmm3 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 7F /r /is4 VFNMSSUBSD xmm0, xmm1, xmm2, xmm3/m64 | V/V | FMA | Multiply scalar double-precision floating-point value in xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |

Description

Performs an SIMD multiply of the low double-precision floating-point value from the second source operand to the low 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 double-precision element to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:64] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMSSUBSD DEST, SRC1, SRC2, SRC3

DEST[63:0] \leftarrow (- (SRC1[63:0]*SRC2[63:0]) - SRC3[63:0])
DEST[255:64] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSSUBSS - Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

| Opcode/ Instruction | Mode Support | CPUID Feature Flag | Description |
|--|-----------------|--------------------------|---|
| VEX.NDS.128.66.0F3A.W0 7E /r /is4 VFNMSSUBSS xmm0, xmm1, xmm2/m32, xmm3 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66.0F3A.W1 7E /r /is4 VFNMSSUBSS xmm0, xmm1, xmm2, xmm3/m32 | V/V | FMA | Multiply scalar single-precision floating-point value in xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |

Description

Performs an SIMD multiply of the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the second 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 single-precision value to the destination operand. An 8-bit immediate field (imm8) encodes first source register in bits[7:4]. Bits[3:0] of imm8 are reserved for future use and must be 0000b.

VEX.128 encoded version: If VEX.W bit is 0, the first source operand (encoded in imm8[7:4]) and third source operand (encoded in VEX.vvvv) are XMM registers, the second source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. If VEX.W bit is 1, the first source operand (encoded in imm8[7:4]) and second source operand (encoded in VEX.vvvv) are XMM registers, the third source operand (encoded in rm_field) is a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits[255:32] of the YMM destination are zeroed.

In 32-bit mode imm8[7] is ignored.

Operation

In the operations below, “-” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding)

VFNMSSUBSS DEST, SRC1, SRC2, SRC3

DEST[31:0] \leftarrow (- (SRC1[31:0]*SRC2[31:0]) - SRC3[31:0])
DEST[255:32] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSSUBSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c, const int ctrl);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

APPENDIX A INSTRUCTION SUMMARY

Most SSE/SSE2/SSE3/SSSE3/SSE4 Instructions have been promoted to support VEX.128 encodings which, for non-memory-store versions implies support for zeroing upper bits of YMM registers. Table A-1 summarizes the promotion status for existing instructions. The column "VEX.256" indicates whether 256-bit vector form of the instruction using the VEX.256 prefix encoding is supported. The column "VEX.128" indicates whether the instruction using VEX.128 prefix encoding is supported.

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? |
|------------------|------------------|----------|-------------|---------------------------|
| yes | yes | YY OF 1X | MOVUPS | |
| no | yes | | MOVSS | scalar |
| yes | yes | | MOVUPD | |
| no | yes | | MOVSD | scalar |
| no | yes | | MOVLPS | Note 1 |
| no | yes | | MOVLPD | Note 1 |
| no | yes | | MOVLHPS | Redundant with VPER-MILPS |
| yes | yes | | MOVDDUP | |
| yes | yes | | MOVSLDUP | |
| yes | yes | | UNPCKLPS | |
| yes | yes | | UNPCKLPD | |
| yes | yes | | UNPCKHPS | |
| yes | yes | | UNPCKHPD | |
| no | yes | | MOVHPS | Note 1 |
| no | yes | | MOVHPD | Note 1 |
| no | yes | | MOVHLPS | Redundant with VPER-MILPS |
| yes | yes | | MOVAPS | |
| yes | yes | | MOVSHDUP | |
| yes | yes | | MOVAPD | |
| no | no | | CVTPI2PS | MMX |
| no | yes | | CVTSI2SS | scalar |
| no | no | | CVTPI2PD | MMX |
| no | yes | | CVTSI2SD | scalar |
| no | yes | | MOVNTPS | |
| no | yes | | MOVNTPD | |
| no | no | | CVTTPS2PI | MMX |
| no | yes | | CVTTSS2SI | scalar |
| no | no | | CVTTPD2PI | MMX |
| no | yes | | CVTTSD2SI | scalar |
| no | no | | CVTPS2PI | MMX |
| no | yes | | CVTSS2SI | scalar |
| no | no | | CVTPD2PI | MMX |
| no | yes | CVTSD2SI | scalar | |

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? |
|---------------------|---------------------|----------|-------------|----------------|
| no | yes | YY OF 5X | UCOMISS | scalar |
| no | yes | | UCOMISD | scalar |
| no | yes | | COMISS | scalar |
| no | yes | | COMISD | scalar |
| yes | yes | | MOVMSKPS | |
| yes | yes | | MOVMSKPD | |
| yes | yes | | SQRTPS | |
| no | yes | | SQRTSS | scalar |
| yes | yes | | SQRTPD | |
| no | yes | | SQRTSD | scalar |
| yes | yes | | RSQRTPS | |
| no | yes | | RSQRTSS | scalar |
| yes | yes | | RCPPS | |
| no | yes | | RCPSS | scalar |
| yes | yes | | ANDPS | |
| yes | yes | | ANDPD | |
| yes | yes | | ANDNPS | |
| yes | yes | | ANDNPD | |
| yes | yes | | ORPS | |
| yes | yes | | ORPD | |
| yes | yes | | XORPS | |
| yes | yes | | XORPD | |
| yes | yes | | ADDPS | |
| no | yes | | ADDSS | scalar |
| yes | yes | | ADDPD | |
| no | yes | | ADDSD | scalar |
| yes | yes | | MULPS | |
| no | yes | | MULSS | scalar |
| yes | yes | | MULPD | |
| no | yes | | MULSD | scalar |
| yes | yes | | CVTTPS2PD | |
| no | yes | | CVTSS2SD | scalar |
| yes | yes | | CVTPD2PS | |
| no | yes | CVTSD2SS | scalar | |
| yes | yes | CVTDQ2PS | | |

INSTRUCTION SUMMARY

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? |
|---------------------|---------------------|----------|-------------|----------------|
| yes | yes | | CVTPS2DQ | |
| yes | yes | | CVTTPS2DQ | |
| yes | yes | | SUBPS | |
| no | yes | | SUBSS | scalar |
| yes | yes | | SUBPD | |
| no | yes | | SUBSD | scalar |
| yes | yes | | MINPS | |
| no | yes | | MINSS | scalar |
| yes | yes | | MINPD | |
| no | yes | | MINSB | scalar |
| yes | yes | | DIVPS | |
| no | yes | | DIVSS | scalar |
| yes | yes | | DIVPD | |
| no | yes | | DIVSD | scalar |
| yes | yes | | MAXPS | |
| no | yes | | MAXSS | scalar |
| yes | yes | | MAXPD | |
| no | yes | | MAXSD | scalar |
| no | yes | YY OF 6X | PUNPCKLBW | VI |
| no | yes | | PUNPCKLWD | VI |
| no | yes | | PUNPCKLDQ | VI |
| no | yes | | PACKSSWB | VI |
| no | yes | | PCMPGTB | VI |
| no | yes | | PCMPGTW | VI |
| no | yes | | PCMPGTD | VI |
| no | yes | | PACKUSWB | VI |
| no | yes | | PUNPCKHBW | VI |
| no | yes | | PUNPCKHWD | VI |
| no | yes | | PUNPCKHDQ | VI |
| no | yes | | PACKSSDW | VI |
| no | yes | | PUNPCKLQDQ | VI |
| no | yes | | PUNPCKHQDQ | VI |
| no | yes | | MOVB | scalar |
| no | yes | | MOVQ | scalar |
| yes | yes | | MOVBQ | |

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? |
|---------------------|---------------------|----------|-------------|----------------|
| yes | yes | | MOVDQU | |
| no | yes | YY OF 7X | PSHUFD | VI |
| no | yes | | PSHUFHW | VI |
| no | yes | | PSHUFLW | VI |
| no | yes | | PCMPEQB | VI |
| no | yes | | PCMPEQW | VI |
| no | yes | | PCMPEQD | VI |
| yes | yes | | HADDDP | |
| yes | yes | | HADDPS | |
| yes | yes | | HSUBPD | |
| yes | yes | | HSUBPS | |
| no | yes | | MOVD | VI |
| no | yes | | MOVQ | VI |
| yes | yes | | MOVDQA | |
| yes | yes | | MOVDQU | |
| no | yes | YY OF AX | LDMXCSR | |
| no | yes | | STMXCSR | |
| yes | yes | YY OF CX | CMPPS | |
| no | yes | | CMPSS | scalar |
| yes | yes | | CMPPD | |
| no | yes | | CMPSD | scalar |
| no | yes | | PINSRW | VI |
| no | yes | | PEXTRW | VI |
| yes | yes | | SHUFPS | |
| yes | yes | | SHUFPD | |
| yes | yes | YY OF DX | ADDSUBPD | |
| yes | yes | | ADDSUBPS | |
| no | yes | | PSRLW | VI |
| no | yes | | PSRLD | VI |
| no | yes | | PSRLQ | VI |
| no | yes | | PADDQ | VI |
| no | yes | | PMULLW | VI |
| no | no | | MOVQ2DQ | MMX |
| no | no | | MOVDQ2Q | MMX |
| no | yes | | PMOVMASKB | VI |

INSTRUCTION SUMMARY

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? |
|---------------------|---------------------|----------|-------------|----------------|
| no | yes | | PSUBUSB | VI |
| no | yes | | PSUBUSW | VI |
| no | yes | | PMINUB | VI |
| no | yes | | PAND | VI |
| no | yes | | PADDUSB | VI |
| no | yes | | PADDUSW | VI |
| no | yes | | PMAXUB | VI |
| no | yes | | PANDN | VI |
| no | yes | YY OF EX | PAVGB | VI |
| no | yes | | PSRAW | VI |
| no | yes | | PSRAD | VI |
| no | yes | | PAVGW | VI |
| no | yes | | PMULHUW | VI |
| no | yes | | PMULHW | VI |
| yes | yes | | CVTPD2DQ | |
| yes | yes | | CVTTPD2DQ | |
| yes | yes | | CVTDQ2PD | |
| no | yes | | MOVNTDQ | VI |
| no | yes | | PSUBSB | VI |
| no | yes | | PSUBSW | VI |
| no | yes | | PMINSW | VI |
| no | yes | | POR | VI |
| no | yes | | PADDSB | VI |
| no | yes | | PADDSW | VI |
| no | yes | | PMAXSW | VI |
| no | yes | | PXOR | VI |
| yes | yes | YY OF FX | LDDQU | VI |
| no | yes | | PSLLW | VI |
| no | yes | | PSLLD | VI |
| no | yes | | PSLLQ | VI |
| no | yes | | PMULUDQ | VI |
| no | yes | | PMADDWD | VI |
| no | yes | | PSADBW | VI |
| no | yes | | MASKMOVDQU | |
| no | yes | | PSUBB | VI |

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? | |
|---------------------|---------------------|-----------|-------------|----------------|--|
| no | yes | SSSE3 | PSUBW | VI | |
| no | yes | | PSUBD | VI | |
| no | yes | | PSUBQ | VI | |
| no | yes | | PADDB | VI | |
| no | yes | | PADDW | VI | |
| no | yes | | PADDD | VI | |
| no | yes | | PHADDW | VI | |
| no | yes | | PHADDSW | VI | |
| no | yes | | PHADDD | VI | |
| no | yes | | PHSUBW | VI | |
| no | yes | | PHSUBSW | VI | |
| no | yes | | PHSUBD | VI | |
| no | yes | | PMADDUBSW | VI | |
| no | yes | | PALIGNR | VI | |
| no | yes | | PSHUFB | VI | |
| no | yes | | PMULHRSW | VI | |
| no | yes | | PSIGNB | VI | |
| no | yes | | PSIGNW | VI | |
| no | yes | | PSIGND | VI | |
| no | yes | | PABSB | VI | |
| no | yes | | PABSW | VI | |
| no | yes | | PABSD | VI | |
| yes | yes | | SSE4.1 | BLENDPS | |
| yes | yes | | | BLENDPD | |
| yes | yes | BLENDVPS | | Note 2 | |
| yes | yes | BLENDVPD | | Note 2 | |
| no | yes | DPPD | | | |
| yes | yes | DPPS | | | |
| no | yes | EXTRACTPS | | Note 3 | |
| no | yes | INSERTPS | | Note 3 | |
| no | yes | MOVNTDQA | | | |
| no | yes | MPSADBW | | VI | |
| no | yes | PACKUSDW | VI | | |
| no | yes | PBLENDVB | VI | | |
| no | yes | PBLENDW | VI | | |

INSTRUCTION SUMMARY

| VEX.256 Encoding | VEX.128 Encoding | group | Instruction | If No, Reason? |
|---------------------|---------------------|--------|-------------|----------------|
| no | yes | | PCMPEQQ | VI |
| no | yes | | PEXTRD | VI |
| no | yes | | PEXTRQ | VI |
| no | yes | | PEXTRB | VI |
| no | yes | | PEXTRW | VI |
| no | yes | | PHMINPOSUW | VI |
| no | yes | | PINSRB | VI |
| no | yes | | PINSRD | VI |
| no | yes | | PINSRQ | VI |
| no | yes | | PMAXSB | VI |
| no | yes | | PMAXSD | VI |
| no | yes | | PMAXUD | VI |
| no | yes | | PMAXUW | VI |
| no | yes | | PMINSB | VI |
| no | yes | | PMINSD | VI |
| no | yes | | PMINUD | VI |
| no | yes | | PMINUW | VI |
| no | yes | | PMOVSXxx | VI |
| no | yes | | PMOVZXxx | VI |
| no | yes | | PMULDQ | VI |
| no | yes | | PMULLD | VI |
| yes | yes | | PTEST | |
| yes | yes | | ROUNDPD | |
| yes | yes | | ROUNDPS | |
| no | yes | | ROUNDSD | scalar |
| no | yes | | ROUNDSS | scalar |
| no | yes | SSE4.2 | PCMPGTQ | VI |
| no | no | SSE4.2 | CRC32c | integer |
| no | yes | | PCMPESTRI | VI |
| no | yes | | PCMPESTRM | VI |
| no | yes | | PCMPISTRI | VI |
| no | yes | | PCMPISTRM | VI |
| no | no | SSE4.2 | POPCNT | integer |

Description of Column "If No, Reason?"

MMX: Instructions referencing MMX registers do not support VEX

Scalar: Scalar instructions are not promoted to 256-bit

integer: integer instructions are not promoted.

VI: "Vector Integer" instructions are not promoted to 256-bit

Note 1: MOVLDPD/PS and MOVHPD/PS are not promoted to 256-bit. The equivalent functionality are provided by VINSERTF128 and VEXTRACTF128 instructions as the existing instructions have no natural 256b extension

Note 2: BLENDVDP and BLENDVPS are superseded by the more flexible VBLENDVDP and VBLENDVPS.

Note 3: It is expected that using 128-bit INSERTPS followed by a VINSERTF128 would be better than promoting INSERTPS to 256-bit (for example).

Table A-2. AVX, FMA and AES New Instructions

| Opcode | Instruction | Description |
|---------------------------|--|---|
| 66 0F 38 DE /r | AESDEC xmm1, xmm2/m128 | Perform 1 round of AES decryption of xmm1 using the 128-bit round key from the xmm2/m128. |
| 66 0F 38 DF /r | AESDECLAST xmm1, xmm2/m128 | Perform the last round of AES decryption of xmm1 using the 128 bit round key from xmm2/m128. |
| VEX.NDS.128.66.0F38 DE /r | VAESDEC xmm1, xmm2, xmm3/m128 | Perform 1 round of AES decryption of xmm2 using the 128-bit round key from the xmm3/m128, and stores the result in xmm1. |
| VEX.NDS.128.66.0F38 DF /r | VAESDECLAST xmm1, xmm2, xmm3/m128 | Perform the last round of AES decryption of xmm2 using the 128 bit round key from xmm3/m128, and stores the result in xmm1. |
| 66 0F 38 DC /r | AESENC xmm1, xmm2/m128 | Perform 1 round of AES encryption of xmm1 using the 128-bit round key from the xmm2/m128. |
| 66 0F 38 DD /r | AESENCLAST xmm1, xmm2/m128 | Perform the last round of AES encryption of xmm1 using the 128 bit round key from xmm2/m128. |
| VEX.NDS.128.66.0F38 DC /r | VAESENC xmm1, xmm2, xmm3/m128 | Perform 1 round of AES encryption of xmm2 using the 128-bit round key from the xmm3/m128, and stores the result in xmm1. |
| VEX.NDS.128.66.0F38 DD /r | VAESENCLAST xmm1, xmm2, xmm3/m128 | Perform the last round of AES encryption of xmm2 using the 128 bit round key from xmm3/m128, and stores the result in xmm1. |
| 66 0F 38 DB /r | AESIMC xmm1, xmm2/m128 | Perform the InvMixColumn operation using xmm2/mem and store result in xmm1 |
| VEX.128.66.0F38 DB /r | VAESIMC xmm1, xmm2/m128 | Perform the InvMixColumn operation using xmm2/mem and store result in xmm1 |
| 66 0F 3A DF /r ib | AESKEYGENASSIST xmm1, xmm2/m128, imm8 | Assist in AES round key generation using an immediate round control byte, a key specified in xmm2/m128 and stores the result in xmm1. |
| VEX.128.66.0F3A DF /r ib | VAESKEYGENASSIST xmm1, xmm2/m128, imm8 | Assist in AES round key generation using an immediate round control byte, a key specified in xmm2/m128 and stores the result in xmm1. |
| VEX.256.66.0F38 1A /r | VBROADCASTF128 ymm1, m128 | Broadcast 128-bit floating-point values in mem to low and high 128-bits in ymm1 |
| VEX.256.66.0F38 19/r | VBROADCASTSD ymm1, m64 | Broadcast double-precision floating-point element in mem to four locations in ymm1 |

| Opcode | Instruction | Description |
|---------------------------------------|---|--|
| VEX.256.66.0F38 18 /r | VBROADCASTSS ymm1, m32 | Broadcast single-precision floating-point element in mem to eight locations in ymm1 |
| VEX.128.66.0F38 18/r | VBROADCASTSS xmm1, m32 | Broadcast single-precision floating-point element in mem to four locations in xmm1 |
| VEX.256.66.0F3A 19 /r ib | VEXTRACTF128 xmm1/m128, ymm2, imm8 | Extracts 128-bits of packed floating-point values from ymm2 and store results in xmm1/mem |
| VEX.NDS.128.66. 0F3A.W0 69 /r /is4 | VFMADDPD xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 69 /r /is4 | VFMADDPD xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 69 /r /is4 | VFMADDPD ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 69 /r /is4 | VFMADDPD ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 68 /r /is4 | VFMADDPS xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 68 /r /is4 | VFMADDPS xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 68 /r /is4 | VFMADDPS ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 68 /r /is4 | VFMADDPS ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 6B /r /is4 | VFMADDSD xmm0, xmm1, xmm2/m64, xmm3 | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 6B /r /is4 | VFMADDSD xmm0, xmm1, xmm2, xmm3/m64 | Multiply scalar double-precision floating-point value in xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 6A /r /is4 | VFMADDSS xmm0, xmm1, xmm2/m32, xmm3 | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, add to xmm3 and put result in xmm0. |

INSTRUCTION SUMMARY

| Opcode | Instruction | Description |
|---------------------------------------|--|--|
| VEX.NDS.128.66. 0F3A.W1 6A /r /is4 | VFMADDSS xmm0, xmm1, xmm2, xmm3/m32 | Multiply scalar single-precision floating-point value in xmm1 and xmm2, add to xmm3/mem and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 5D /r /is4 | VFMADDSUBPD xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 5D /r /is4 | VFMADDSUBPD xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 5D /r /is4 | VFMADDSUBPD ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 5D /r /is4 | VFMADDSUBPD ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 5C /r /is4 | VFMADDSUBPS xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 5C /r /is4 | VFMADDSUBPS xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 5C /r /is4 | VFMADDSUBPS ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 5C /r /is4 | VFMADDSUBPS ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 5F /r /is4 | VFMSUBADDPD xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 5F /r /is4 | VFMSUBADDPD xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm0. |

| Opcode | Instruction | Description |
|---------------------------------------|--|--|
| VEX.NDS.256.66. 0F3A.W0 5F /r /is4 | VFMSUBADDPD ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 5F /r /is4 | VFMSUBADDPD ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 5E /r /is4 | VFMSUBADDPS xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 5E /r /is4 | VFMSUBADDPS xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 5E /r /is4 | VFMSUBADDPS ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 5E /r /is4 | VFMSUBADDPS ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 6D /r /is4 | VFMSUBPD xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 6D /r /is4 | VFMSUBPD xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 6D /r /is4 | VFMSUBPD ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 6D /r /is4 | VFMSUBPD ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 6C /r /is4 | VFMSUBPS xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 6C /r /is4 | VFMSUBPS xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |

| Opcode | Instruction | Description |
|---------------------------------------|--|--|
| VEX.NDS.256.66. 0F3A.W0 6C /r /is4 | VFMSUBPS ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm3 and put result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 6C /r /is4 | VFMSUBPS ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 6F /r /is4 | VFMSUBSD xmm0, xmm1, xmm2/m64, xmm3 | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 6F /r /is4 | VFMSUBSD xmm0, xmm1, xmm2, xmm3/m64 | Multiply scalar double-precision floating-point value in xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 6E /r /is4 | VFMSUBSS xmm0, xmm1, xmm2/m32, xmm3 | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, subtract xmm3 and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 6E /r /is4 | VFMSUBSS xmm0, xmm1, xmm2, xmm3/m32 | Multiply scalar single-precision floating-point value in xmm1 and xmm2, subtract xmm3/mem and put result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 79 /r /is4 | VFNMADDPD xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 79 /r /is4 | VFNMADDPD xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 79 /r /is4 | VFNMADDPD ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm3. Put the result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 79 /r /is4 | VFNMADDPD ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem. Put the result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 78 /r /is4 | VFNMADDPS xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 78 /r /is4 | VFNMADDPS xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |

| Opcode | Instruction | Description |
|---------------------------------------|--|--|
| VEX.NDS.256.66. 0F3A.W0 78 /r /is4 | VFNMADDPS ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm3. Put the result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 78 /r /is4 | VFNMADDPS ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem. Put the result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 7B /r /is4 | VFNMADDSD xmm0, xmm1, xmm2/m64, xmm3 | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 7B /r /is4 | VFNMADDSD xmm0, xmm1, xmm2, xmm3/m64 | Multiply scalar double-precision floating-point value in xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 7A /r /is4 | VFNMADDSS xmm0, xmm1, xmm2/m32, xmm3 | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 7A /r /is4 | VFNMADDSS xmm0, xmm1, xmm2, xmm3/m32 | Multiply scalar single-precision floating-point value in xmm1 and xmm2, negate the multiplication result and add to xmm3/mem. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 7D /r /is4 | VFNMSUBPD xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 7D /r /is4 | VFNMSUBPD xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 7D /r /is4 | VFNMSUBPD ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm3. Put the result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 7D /r /is4 | VFNMSUBPD ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem. Put the result in ymm0. |

INSTRUCTION SUMMARY

| Opcode | Instruction | Description |
|---------------------------------------|--|--|
| VEX.NDS.128.66. 0F3A.W0 7C /r /is4 | VFNMSUBPS xmm0, xmm1, xmm2/m128, xmm3 | Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 7C /r /is4 | VFNMSUBPS xmm0, xmm1, xmm2, xmm3/m128 | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66. 0F3A.W0 7C /r /is4 | VFNMSUBPS ymm0, ymm1, ymm2/m256, ymm3 | Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm3. Put the result in ymm0. |
| VEX.NDS.256.66. 0F3A.W1 7C /r /is4 | VFNMSUBPS ymm0, ymm1, ymm2, ymm3/m256 | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem. Put the result in ymm0. |
| VEX.NDS.128.66. 0F3A.W0 7F /r /is4 | VFNMSUBSD xmm0, xmm1, xmm2/m64, xmm3 | Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 7F /r /is4 | VFNMSUBSD xmm0, xmm1, xmm2, xmm3/m64 | Multiply scalar double-precision floating-point value in xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W0 7E /r /is4 | VFNMSUBSS xmm0, xmm1, xmm2/m32, xmm3 | Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm3. Put the result in xmm0. |
| VEX.NDS.128.66. 0F3A.W1 7E /r /is4 | VFNMSUBSS xmm0, xmm1, xmm2, xmm3/m32 | Multiply scalar single-precision floating-point value in xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem. Put the result in xmm0. |
| VEX.NDS.256.66. 0F3A 18 /r ib | VINSERTF128 ymm1, ymm2, xmm3/m128, imm8 | Insert 128-bits of packed floating-point values from xmm3/mem and the remaining values from ymm2 into ymm1 |
| VEX.NDS.128.66. 0F38 2C /r | VMASKMOVPS xmm1, xmm2, m128 | Load packed single-precision values from mem using mask in xmm2 and store in xmm1 |
| VEX.NDS.256.66. 0F38 2C /r | VMASKMOVPS ymm1, ymm2, m256 | Load packed single-precision values from mem using mask in ymm2 and store in ymm1 |

| Opcode | Instruction | Description |
|----------------------------------|---|--|
| VEX.NDS.128.66. 0F38 2D/r | VMASKMOVPD xmm1, xmm2, m128 | Load packed double-precision values from mem using mask in xmm2 and store in xmm1 |
| VEX.NDS.256.66. 0F38 2D /r | VMASKMOVPD ymm1, ymm2, m256 | Load packed double-precision values from mem using mask in ymm2 and store in ymm1 |
| VEX.NDS.128.66. 0F38 2E /r | VMASKMOVPS m128, xmm1, xmm2 | Store packed single-precision values from xmm2 using mask in xmm1 |
| VEX.NDS.256.66. 0F38 2E /r | VMASKMOVPS m256, ymm1, ymm2 | Store packed single-precision values from ymm2 mask in ymm1 |
| VEX.NDS.128.66. 0F38 2F /r | VMASKMOVPD m128, xmm1, xmm2 | Store packed double-precision values from xmm2 using mask in xmm1 |
| VEX.NDS.256.66. 0F38 2F /r | VMASKMOVPD m256, ymm1, ymm2 | Store packed double-precision values from ymm2 using mask in ymm1 |
| VEX.NDS.128.66. 0F38 0D /r | VPERMILPD xmm1, xmm2, XMM3/m128 | Permute Double-Precision Floating-Point values in xmm2 using controls from xmm3/mem and store result in xmm1 |
| VEX.128.66.0F3A 05 /r ib | VPERMILPD xmm1, xmm2/m128, imm8 | Permute Double-Precision Floating-Point values in xmm2/mem using controls from imm8 and store result in xmm1 |
| VEX.NDS.256.66. 0F38 0D /r | VPERMILPD ymm1, ymm2, ymm3/m256 | Permute Double-Precision Floating-Point values in ymm2 using controls from xmm3/mem and store result in ymm1 |
| VEX.256.66.0F3A 05 /r ib | VPERMILPD ymm1, ymm2/m256 imm8 | Permute Double-Precision Floating-Point values in ymm2/mem using controls from imm8 and store result in ymm1 |
| VEX.NDS.128.66. 0F38 0C /r | VPERMILPS xmm1, xmm2, xmm3/m128 | Permute Single-Precision Floating-Point values in xmm2 using controls from xmm3/mem and store result in xmm1 |
| VEX.128.66.0F3A 04 /r ib | VPERMILPS xmm1, xmm2/m128, imm8 | Permute Single-Precision Floating-Point values in xmm2/mem using controls from imm8 and store result in xmm1 |
| VEX.NDS.256.66. 0F38 0C /r | VPERMILPS ymm1, ymm2, YMM/m256 | Permute Single-Precision Floating-Point values in ymm2 using controls from ymm3/mem and store result in ymm1 |
| VEX.256.66.0F3A 04 /r ib | VPERMILPS ymm1, ymm2/m256, imm8 | Permute Single-Precision Floating-Point values in ymm2/mem using controls from imm8 and store result in ymm1 |
| VEX.NDS.256.66. 0F3A 06 /r ib | VPERM2F128 ymm1, ymm2, ymm3/m256, imm8 | Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1 |

INSTRUCTION SUMMARY

| Opcode | Instruction | Description |
|---------------------------------------|---|--|
| VEX.NDS.128.66. 0F3A.W0 49 /r /is4 | VPERMIL2PD xmm1, xmm2, xmm3/m128, xmm4, imz2 | Permute Double-Precision Floating-Point values in xmm2 and xmm3/mem using controls from xmm4, the results are stored in xmm1 with selective zero-match controls. |
| VEX.NDS.128.66. 0F3A.W1 49 /r /is4 | VPERMIL2PD xmm1, xmm2, xmm3, xmm4/m128, imz2 | Permute Double-Precision Floating-Point values in xmm2 and xmm3 using controls from xmm4/mem, the results are stored in xmm1 with selective zero-match controls. |
| VEX.NDS.256.66. 0F3A.W0 49 /is4 | VPERMIL2PD ymm1, ymm2, ymm3/m256, ymm4, imz2 | Permute Double-Precision Floating-Point values in ymm2 and ymm3/mem using controls from ymm4, the results are stored in ymm1 with selective zero-match controls. |
| VEX.NDS.256.66. 0F3A.W1 49 /is4 | VPERMIL2PD ymm1, ymm2, ymm3, ymm4/m256, imz2 | Permute Double-Precision Floating-Point values in ymm2 and ymm3 using controls from ymm4/mem. the results are stored in ymm1 with selective zero-match controls. |
| VEX.NDS.128.66. 0F3A.W0 48 /r /is4 | VPERMIL2PS xmm1, xmm2, xmm3/m128, xmm4, imz2 | Permute Single-Precision Floating-Point values in xmm2 and xmm3/mem using controls from xmm4, the results are stored in xmm1 with selective zero-match controls. |
| VEX.NDS.128.66. 0F3A.W1 48 /r /is4 | VPERMIL2PS xmm1, xmm2, xmm3, xmm4/m128, imz2 | Permute Single-Precision Floating-Point values in xmm2 and xmm3 using controls from xmm4/mem, the results are stored in xmm1 with selective zero-match controls. |
| VEX.NDS.256.66. 0F3A.W0 48 /is4 | VPERMIL2PS ymm1, ymm2, ymm3/m256, ymm4, imz2 | Permute Single-Precision Floating-Point values in ymm2 and ymm3/mem using controls from ymm4, the results are stored in ymm1 with selective zero-match controls. |
| VEX.NDS.256.66. 0F3A.W1 48 /is4 | VPERMIL2PS ymm1, ymm2, ymm3, ymm4/m256, imz2 | Permute Single-Precision Floating-Point values in ymm2 and ymm3 using controls from ymm4/mem, the results are stored in ymm1 with selective zero-match controls. |
| 66 0F 3A 44 /r ib | PCLMULQDQ xmm1, xmm2/m128, imm8 | Carry-less multiplication of a pair of quadword selected by an immediate byte from xmm2/m128 and xmm1, stores the 128-bit result in xmm1. |
| VEX.256.66.0F38 0E /r | VTESTPS ymm1, ymm2/m256 | Set ZF if ymm2/mem AND ymm1 result is all 0s in packed single-precision sign bits. Set CF if ymm2/mem AND NOT ymm1 result is all 0s in packed single-precision sign bits. |

| Opcode | Instruction | Description |
|--------------------------|----------------------------|--|
| VEX.256.66.0F38 0F /r | VTESTPD ymm1, ymm2/m256 | Set ZF if ymm2/mem AND ymm1 result is all 0s in packed double-precision sign bits. Set CF if ymm2/mem AND NOT ymm1 result is all 0s in packed double-precision sign bits. |
| VEX.128.66.0F38 0E /r | VTESTPS xmm1, xmm2/m128 | Set ZF if xmm2/mem AND xmm1 result is all 0s in packed single-precision sign bits. Set CF if xmm2/mem AND NOT xmm1 result is all 0s in packed single-precision sign bits. |
| VEX.128.66.0F38 0F /r | VTESTPD xmm1, xmm2/m128 | Set ZF if xmm2/mem AND xmm1 result is all 0s in packed single precision sign bits. Set CF if xmm2/mem AND NOT xmm1 result is all 0s in packed double-precision sign bits. |
| VEX.256.0F 77 | VZEROALL | Zero all YMM registers |
| VEX.128.0F 77 | VZERoupper | Zero upper 128 bits of all YMM registers |

APPENDIX B INSTRUCTION OPCODE MAP

GREEN cells are existing instructions promoted to VEX.128

BLUE cells are existing instructions promoted to VEX.256 and VEX.128

RED cells are AVX and FMA new instructions

YELLOW cells are Non-VEX encoded new instructions

OF

| OF | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F | |
|----|--------------------|------------------|--------------------------------|----------------------|-------------------|--------------------|--------------------------------|--------------------------------|-------------------|--------------------|-------------------|-------------------|-------------------|------------------|-------------------|-----------------|
| 0 | grp6 | grp7 | rar Ev, Ew | rar Ev, Ew | | syscall | cls | sysret | invd | wbinvd | | ud2 | | NOP Ev | | |
| 1 | movups Vps, Wps | movups Vps, Wps | movlps Vq, Uq movlps Vq, Uq | movlps Mq, Vq | unpcklps Vps, Wq | unpcklps Vps, Wq | movhps Vq, Uq movhps Mq, Vq | movhps Mq, Vq | prefetch | | | | | | Nop Ev | |
| 2 | mov Rd, Cd | mov Rd, Dd | mov Cd, Rd | mov Dd, Rd | | | | | movaps Vps, Wps | movaps Vps, Wps | cvtdq2ps Vps, Cq | movmps Mps, Vps | cvtps2pi Cq, Wps | cvtps2pi Cq, Wps | ucomiss Vss, Wss | comiss Vss, Wss |
| 3 | wmsr | rdtsc | rdmsr | rdpmc | sysenter | sysexit | | | 3 byte escape | | | | | | | |
| 4 | cmovo Gv, Ev | cmovno Gv, Ev | cmovb/oha nb Gv, Ev | cmovae/nca nb Gv, Ev | cmovnz Gv, Ev | cmovne/rz Gv, Ev | cmovbe/na Gv, Ev | cmovnl/be Gv, Ev | cmovs Gv, Ev | cmovns Gv, Ev | cmovp/pe Gv, Ev | cmovpl/pe Gv, Ev | cmovl/rnge Gv, Ev | cmovnl/ge Gv, Ev | cmovle/rng Gv, Ev | cmovnl/g Gv, Ev |
| 5 | movmskps Gd, Ups | scrtps Vps, Wps | rsqrtps Vps, Wps | rcpps Vps, Wps | andps Vps, Wps | andps Vps, Wps | orps Vps, Wps | xorps Vps, Wps | addps Vps, Wps | mulps Vps, Wps | cvtdq2pd Vps, Wps | cvtdq2ps Vps, Wps | subps Vps, Wps | minps Vps, Wps | divps Vps, Wps | maxps Vps, Wps |
| 6 | punpcklwb Pq, Qd | punpcklwd Pq, Qd | punpckldq Pq, Qd | packsswb Pq, Qq | pcmpgltb Pq, Qq | pcmpgtw Pq, Qq | pcmpgtd Pq, Qq | packuswb Pq, Qq | punpckhbw Pq, Qd | punpckhwd Pq, Qd | punpckhdq Pq, Qd | packssdwb Pq, Qd | | | movd/q Pd, Ed/q | movq Pq, Qq |
| 7 | psuhfwb Pq, Qq, lb | grp12 | grp13 | grp14 | pcmpeqwb Pq, Qq | pcmpeqw Pq, Qq | pcmpeqd Pq, Qq | emms/Vzer super/Vzer sal | vmread Ed/q, Gd/q | vmwrite Gd/q, Ed/q | | | | | movd/q Ed/q, Pd | movq Cq, Pq |
| 8 | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) | jcc (long) |
| 9 | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc | setcc |
| A | push FS | pop FS | cpuid | bt Ev, Gv | shrd Ev, Gv, lb | shrd Ev, Gv, Ql | | movzx Gv, Eb | movzx Gv, Ew | push GS | pop GS | rsm | bts Ev, Gv | shrd Ev, Gv, lb | shrd Ev, Gv, Ql | Grp15 |
| B | cmprchg Eb, Gb | cmprchg Ev, Gv | lss Gv, Mp | btr Ev, Gv | lfs Gv, Mp | lgs Gv, Mp | movzx Gv, Eb | movzx Gv, Ew | jmpbe | grp10 | grp8 | bts Ev, Gv | bsf Ev, Gv | bsr Ev, Gv | movsx Gv, Eb | movsx Gv, Ew |
| C | xadd Eb, Gb | xadd Ev, Gv | cmpps Vps, Wps, lb | movnti Mq, Cd/q | pintrw Pq, Ew, lb | peextrw Gd, Nq, lb | shlps Vps, Wps, lb | grp9 | bswap | bswap | bswap | bswap | bswap | bswap | bswap | bswap |
| D | | psrlw Pq, Qq | psrld Pq, Qq | psrlq Pq, Qq | paddd Pq, Qq | pmullw Pq, Qq | | pmovmskb Gd, Nq | psubusb Pq, Qq | psubusb Pq, Qq | pmminub Pq, Qq | psand Pq, Qq | paddusb Pq, Qq | paddusw Pq, Qq | pmxsub Pq, Qq | pmshl Pq, Qq |
| E | | psavgb Pq, Qq | psraw Pq, Qq | psrad Pq, Qq | psavgw Pq, Qq | pmulhw Pq, Qq | | movntq Mq, Pq | psubsb Pq, Qq | psubsw Pq, Qq | pmminsw Pq, Qq | psor Pq, Qq | paddsb Pq, Qq | paddsw Pq, Qq | pmxasw Pq, Qq | pxor Pq, Qq |
| F | | psllw Pq, Qq | pslld Pq, Qq | psllq Pq, Qq | pmuludq Pq, Qq | pmaddwd Pq, Qq | psadwb Pq, Qq | maskmovq Pq, Nq | psubb Pq, Qq | psubw Pq, Qq | psubd Pq, Qq | psubbq Pq, Qq | paddb Pq, Qq | paddw Pq, Qq | paddq Pq, Qq | |

66 OF

INSTRUCTION OPCODE MAP

| 66 0F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F | |
|-------|-----------------------|-----------------------|--------------------------|----------------------|----------------------|-------------------------|---------------------------|----------------------------|----------------------|----------------------|----------------------|----------------------|-----------------------|-------------------------|---------------------|--------------------|
| 0 | | | | | | | | | | | | | | | | |
| 1 | movupd Vpd, Wpd | movupd Wpd, Vpd | movpd Vq, Mq | movpd Mq, Vq | unpckhpd Vpd, Wq | unpckhpd Vpd, Wq | movhpd Vq, Mq | movhpd Mq, Vq | | | | | | | | |
| 2 | | | | | | | | | movapd Vpd, Wpd | movapd Wpd, Vpd | cvtpd2pd Vpd, Cq | movnpsd Mpd, Vpd | cvtpd2pd Cdq, Wpd | cvtpd2pd Cdq, Wpd | ucomisd Vsd, Wsd | comisd Vsd, Wsd |
| 3 | | | | | | | | | 3 byte escape | | 3 byte escape | | | | | |
| 4 | | | | | | | | | | | | | | | | |
| 5 | movmepd Gd, Upd | scrtpd Vpd, Wpd | | | andpd Vpd, Wpd | andhpd Vpd, Wpd | orpd Vpd, Wpd | xorpd Vpd, Wpd | addpd Vpd, Wpd | mulpd Vpd, Wpd | cvtpd2ps Vps, Wp | cvtps2dq Vdq, Wps | subpd Vpd, Wpd | minpd Vpd, Wpd | divpd Vpd, Wpd | maxpd Vpd, Wpd |
| 6 | punpckhbw Vdq, Wdq | punpckhwd Vdq, Wdq | punpckldq Vdq, Wdq | packsswb Vdq, Wdq | packgtb Vdq, Wdq | packgtw Vdq, Wdq | packgtl Vdq, Wdq | packuswb Vdq, Wdq | punpckhbw Pd, Cdq | punpckhwd Pd, Cdq | punpckhdq Pd, Cdq | packssdwb Pd, Cdq | punpckldq Vdq, Wdq | punpckhdq q Vdq, Wdq | movdq Vdq, Edq | movdqa Vdq, Wdq |
| 7 | psufl Vdq, Wdq, lb | grp12 | grp13 | grp14 | pmpeqb Vdq, Wdq | pmpeqw Vdq, Wdq | pmpeql Vdq, Wdq | | | | | | haddpd Vpd, Wpd | fsuubpd Vpd, Wpd | movdq Edq, Vdq | movdqa Edq, Vdq |
| 8 | | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | | |
| C | | | cmppd Vpd, Wpd, lb | | pinsw Vdq, Ew, lb | pesth Gd, Udq, lb | shufpd Vpd, Wpd, lb | | | | | | | | | |
| D | addsubpd Vpd, Wpd | psrlw Vdq, Wdq | psrlq Vdq, Wdq | psrlq Vdq, Wdq | psrlq Vdq, Wdq | pmulw Vdq, Wdq | movq Wq, Vq | pmovmskb Gd, Udq | psubsw Vdq, Wdq | psubsw Vdq, Wdq | pmulb Vdq, Wdq | psand Vdq, Wdq | psadbw Vdq, Wdq | psadbw Vdq, Wdq | pmrsub Vdq, Wdq | psandq Vdq, Wdq |
| E | psavg Vdq, Wdq | psrav Vdq, Wdq | psrad Vdq, Wdq | psavg Vdq, Wdq | pmulhw Vdq, Wdq | pmulhw Vdq, Wdq | cvtpd2dq Vdq, Wpd | movntdq Mdq, Vdq | psubb Vdq, Wdq | psubsw Vdq, Wdq | pmrsw Vdq, Wdq | psarl Vdq, Wdq | psadbw Vdq, Wdq | psadbw Vdq, Wdq | pmrsw Vdq, Wdq | psarl Vdq, Wdq |
| F | | psllw Vdq, Wdq | pslld Vdq, Wdq | psllq Vdq, Wdq | pmuludq Vdq, Wdq | psradwd Vdq, Wdq | psadbw Vdq, Wdq | maskmovd qu Vdq, Udq | psubb Vdq, Wdq | psubw Vdq, Wdq | psubd Vdq, Wdq | psubq Vdq, Wdq | psadbw Vdq, Wdq | psadbw Vdq, Wdq | psadbw Vdq, Wdq | |

F2 OF

| F2 0F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F |
|-------|------------------------|--------------------|---------------------------|---|---|---|-------------------|----------------------|-------------------|----------------------|----------------------|---------------------|----------------------|-------------------|-------------------|
| 0 | | | | | | | | | | | | | | | |
| 1 | movsd Vsd, Wsd | movsd Vsd, Wsd | movdqup Vq, Wq | | | | | | | | | | | | |
| 2 | | | | | | | | | | cvtsd2sq Vsd, Edq | | cvtsd2sq Gd, Wsd | cvtsd2sq Gdq, Wsd | | |
| 3 | | | | | | | | | 3 byte escape | 3 byte escape | | | | | |
| 4 | | | | | | | | | | | | | | | |
| 5 | | scrtsd Vsd, Wsd | | | | | | | andsd Vsd, Wsd | mulsd Vsd, Wsd | cvtsd2ss Vsd, Wsd | subsd Vsd, Wsd | minsd Vsd, Wsd | divsd Vsd, Wsd | maxsd Vsd, Wsd |
| 6 | | | | | | | | | | | | | | | |
| 7 | psrllw Vdq, Wdq, lb | | | | | | | | | | | haddps Vps, Wps | fsuubps Vps, Wps | x | |
| 8 | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | |
| C | | | cmppsd Vsd, Wsd, lb | | | | | | | | | | | | |
| D | addsubps Vps, Wps | | | | | | movsd2q Pd, Uq | | | | | | | | |
| E | | | | | | | | cvtps2dq Vdq, Wpd | | | | | | | |
| F | psrlq Vdq, Wdq | | | | | | | | | | | | | | |

F3 OF

INSTRUCTION OPCODE MAP

| F3 0F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F |
|-------|----------------|----------------|--------------------|---|---|---|--------------------|---|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|
| 0 | | | | | | | | | | | | | | | |
| 1 | movs Vss, Vss | movs Vss, Vss | movsdup Vq, Wq | | | | movsdup Vq, Wq | | | | | | | | |
| 2 | | | | | | | | | | psrls Vss, Esq | | psrls Sst, Wss | psrls Csq, Wss | | |
| 3 | | | | | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | |
| 5 | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | | | | | | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss |
| 6 | | | | | | | | | | | | | | | movdqu Vdq, Wdq |
| 7 | psrls Vss, Vss | | | | | | | | | | | | | movq Vq, Wq | movdqu Vdq, Wdq |
| 8 | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | |
| B | | | | | | | | | POPCNT | | | | | | |
| C | | | psrls Vss, Vss, Rb | | | | | | | | | | | | |
| D | | | | | | | movq2dq Vdq, Wq | | | | | | | | |
| E | | | | | | | psrls Vss, Vss, Rb | | | | | | | | |
| F | | | | | | | | | | | | | | | |

66 OF 38

| 66 0F 38 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F |
|----------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss |
| 1 | psrls Vss, Vss | | | | | | | | | | | | | | |
| 2 | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | | | | | | | | | | |
| 3 | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | | | | | | | | | | |
| 4 | psrls Vss, Vss | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | |
| C | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss |
| E | | | | | | | | | | | | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss |
| F | | | | | | | | | | | | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss | psrls Vss, Vss |

INSTRUCTION OPCODE MAP

OF 38

| OF3B | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F |
|------|---------------|---------------|---------------|------------------|---------------------|---------------|---------------|------------------|---------------|---------------|------------------|---------------|---------------|---------------|---|
| 0 | prulr-q Cq | prulw-q Cq | prulr-q Cq | prulw-q Fq Cq | prulr-q Vq Fq Cq | prulw-q Cq | prulr-q Cq | prulw-q Fq Cq | prulr-q Cq | prulw-q Cq | prulr-q Fq Cq | | | | |
| 1 | | | | | | | | | | | | prulr-q Cq | | | |
| 2 | | | | | | | | | | | | | prulw-q Cq | | |
| 3 | | | | | | | | | | | | | | prulr-q Cq | |
| 4 | | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | |
| C | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | | | | |
| E | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | |

F2 OF 38

| F2OF3B | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F |
|--------|---------------|---------------|---|---|---|---|---|---|---|----|---|---|---|---|---|
| 0 | | | | | | | | | | | | | | | |
| 1 | | | | | | | | | | | | | | | |
| 2 | | | | | | | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | |
| C | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | | | | |
| E | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | |
| | grulw-q Ev | grulw-q Ev | | | | | | | | | | | | | |

66 OF 3A

INSTRUCTION OPCODE MAP

| 66 0F 3A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F | |
|----------|-----------------------------|----------|-----------|---------|-----------|-----------|------------|-----------|------------|--------------|----------|-------------|-------------|-------------|-------------|-----------------------------|
| 0 | | | | | Vpermilps | Vpermilpd | vperm2f128 | | roundps | roundpd | roundss | roundsd | blendps | blendpd | pblendw | paalignr Vdq, Wdq, lb |
| 1 | | | | | pextrb | pxtrw | pxtrq | extractps | vmsert128 | vextractf128 | | | | | | |
| 2 | pinsrb | insertps | pinsrdq | | | | | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | | |
| 4 | dpps | pppd | mpsadbw | | pcmluq | | | | vpermil2ps | vpermil2pd | vblendps | vblendpd | vblendub | | | |
| 5 | | | | | | | | | | | | vfmaddsubps | vfmaddsubpd | vfmsubaddps | vfmsubaddpd | |
| 6 | pcmpesqm Vdq, Wdq, lb | pcmpesrb | pcmpesrdq | pcmpesq | | | | | vfmaddps | vfmaddpd | vfmaddss | vfmaddsd | vfmsubps | vfmsubpd | vfmsubss | vfmsubsd |
| 7 | | | | | | | | | vfmaddps | vfmaddpd | vfmaddss | vfmaddsd | vfmsubps | vfmsubpd | vfmsubss | vfmsubsd |
| 8 | | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | | |
| C | | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | | | | | aeskeygen assist |
| E | | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | | |

OF 3A

| OF 3A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9A | B | C | D | E | F | |
|-------|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---------------------|
| 0 | | | | | | | | | | | | | | | | paalignr Cdq, lb |
| 1 | | | | | | | | | | | | | | | | |
| 2 | | | | | | | | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | | |
| C | | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | | | | | |
| E | | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | | |

This page was
intentionally left
blank.

A

| | |
|---|------|
| ADDPD - Add Packed Double Precision Floating-Point Values | 5-14 |
| ADDPS- Add Packed Single Precision Floating-Point Values | 5-16 |
| ADDSD- Add Scalar Double Precision Floating-Point Values | 5-18 |
| ADDSS- Add Scalar Single Precision Floating-Point Values | 5-20 |
| ADDSUBPD- Packed Double FP Add/Subtract | 5-22 |
| ADDSUBPS- Packed Single FP Add/Subtract | 5-24 |
| AESDEC/AESDECLAST- Perform One Round of an AES Decryption Flow | 5-28 |
| AESENC/AESENCLAST- Perform One Round of an AES Encryption Flow | 5-26 |
| AESIMC- Perform the AES InvMixColumn Transformation | 5-30 |
| AESKEYGENASSIST - AES Round Key Generation Assist. | 5-31 |
| ANDNPD- Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values .. | 5-37 |
| ANDNPS- Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values .. | 5-39 |
| ANDPD- Bitwise Logical AND of Packed Double Precision Floating-Point Values | 5-33 |
| ANDPS- Bitwise Logical AND of Packed Single Precision Floating-Point Values | 5-35 |

B

| | |
|---|------|
| BLENDPD- Blend Packed Double Precision Floating-Point Values | 5-41 |
| BLENDPS- Blend Packed Single Precision Floating-Point Values | 5-43 |
| BLENDVPD- Blend Packed Double Precision Floating-Point Values | 5-46 |
| BLENDVPS- Blend Packed Single Precision Floating-Point Values | 5-49 |
| Brand information | 2-58 |
| processor brand index | 2-61 |
| processor brand string | 2-58 |

C

| | |
|---|------|
| Cache and TLB information | 2-53 |
| Cache Inclusiveness | 2-37 |
| CLFLUSH instruction | |
| CUID flag | 2-52 |
| CMOVcc flag | 2-52 |
| CMOVcc instructions | |
| CUID flag | 2-52 |
| CMPPD- Compare Packed Double-Precision Floating-Point Values | 5-56 |
| CMPPS- Compare Packed Single-Precision Floating-Point Values | 5-64 |
| CMPSD- Compare Scalar Double-Precision Floating-Point Values | 5-71 |
| CMPSS- Compare Scalar Single-Precision Floating-Point Values | 5-76 |
| CMPXCHG16B instruction | |
| CUID bit | 2-48 |
| CMPXCHG8B instruction | |
| CUID flag | 2-51 |
| COMISD- Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS .. | 5-81 |
| COMISS- Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS .. | 5-83 |

- CPUID instruction 2-34, 2-52
 - 36-bit page size extension 2-52
 - APIC on-chip 2-51
 - basic CPUID information 2-35
 - cache and TLB characteristics 2-35, 2-53
 - CLFLUSH flag 2-52
 - CLFLUSH instruction cache line size 2-46
 - CMPXCHG16B flag 2-48
 - CMPXCHG8B flag 2-51
 - CPL qualified debug store 2-48
 - debug extensions, CR4.DE 2-51
 - debug store supported 2-52
 - deterministic cache parameters leaf 2-36, 2-39, 2-40, 2-41
 - extended function information 2-41
 - feature information 2-50
 - FPU on-chip 2-51
 - FSAVE flag 2-52
 - FXRSTOR flag 2-52
 - HT technology flag 2-53
 - IA-32e mode available 2-42
 - input limits for EAX 2-43
 - L1 Context ID 2-48
 - local APIC physical ID 2-46
 - machine check architecture 2-52
 - machine check exception 2-51
 - memory type range registers 2-51
 - MONITOR feature information 2-57
 - MONITOR/MWAIT flag 2-48
 - MONITOR/MWAIT leaf 2-37, 2-38, 2-39
 - MWAIT feature information 2-57
 - page attribute table 2-52
 - page size extension 2-51
 - performance monitoring features 2-57
 - physical address bits 2-43
 - physical address extension 2-51
 - power management 2-57, 2-58
 - processor brand index 2-46, 2-58
 - processor brand string 2-42, 2-58
 - processor serial number 2-35, 2-52
 - processor type field 2-45
 - PTE global bit 2-52
 - RDMSR flag 2-51
 - returned in EBX 2-46
 - returned in ECX & EDX 2-46
 - self snoop 2-53
 - SpeedStep technology 2-48
 - SS2 extensions flag 2-53
 - SSE extensions flag 2-53
 - SSE3 extensions flag 2-47
 - SSSE3 extensions flag 2-48
 - SYSENTER flag 2-51
 - SYSEXIT flag 2-51
 - thermal management 2-57, 2-58
 - thermal monitor 2-48, 2-52, 2-53
 - time stamp counter 2-51
 - using CPUID 2-34
 - vendor ID string 2-43
 - version information 2-35, 2-57

| | |
|---|-------|
| virtual 8086 Mode flag | 2-51 |
| virtual address bits | 2-43 |
| WRMSR flag | 2-51 |
| CVTDQ2PD- Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values | 5-85 |
| CVTDQ2PS- Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values | 5-87 |
| CVTPD2DQ- Convert Packed Double-Precision Floating-point values to Packed Doubleword Integers | 5-89 |
| CVTPD2PS- Convert Packed Double-Precision Floating-point values to Packed Single-Precision Floating-Point Values | 5-92 |
| CVTPS2DQ- Convert Packed Single Precision Floating-Point Values to Packed Singed Doubleword Integer Values | 5-95 |
| CVTPS2PD- Convert Packed Single Precision Floating-point values to Packed Double Precision Floating-Point Values | 5-97 |
| CVTSD2SI- Convert Scalar Double-Precision Floating-Point Value to Doubleword Intege | 5-100 |
| CVTSD2SS- Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value | 5-102 |
| CVTSI2SD- Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value | 5-104 |
| CVTSI2SS- Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value | 5-106 |
| CVTSS2SD- Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value | 5-108 |
| CVTSS2SI- Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer | 5-110 |
| CVTTPD2DQ- Convert with Truncation Packed Double-Precision Floating-point values to Packed Doubleword Integers | 5-112 |
| CVTTPS2DQ- Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Doubleword Integer Values | 5-115 |
| CVTTSD2SI- Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer | 5-117 |
| CVTTSS2SI- Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer | 5-119 |

D

| | |
|--|-------|
| DIVPD- Divide Packed Double-Precision Floating-Point Values | 5-121 |
| DIVPS- Divide Packed Single-Precision Floating-Point Values | 5-123 |
| DIVSD- Divide Scalar Double-Precision Floating-Point Values | 5-125 |
| DIVSS- Divide Scalar Single-Precision Floating-Point Values | 5-127 |
| DPPD- Dot Product of Packed Double-Precision Floating-Point Values | 5-129 |
| DPPS- Dot Product of Packed Single-Precision Floating-Point Values | 5-131 |

E

| | |
|---|-------|
| EXTRACTPS- Extract packed floating-point values | 5-136 |
|---|-------|

F

| | |
|--------------------------------|------------|
| Feature information, processor | 2-34 |
| FMA operation | 2-12, 2-13 |
| FXRSTOR instruction | |
| CPUID flag | 2-52 |
| FXSAVE instruction | |
| CPUID flag | 2-52 |

H

| | |
|--|-------|
| HADDDPD- Add Horizontal Double Precision Floating-Point Values | 5-138 |
| HADDPS- Add Horizontal Single Precision Floating-Point Values | 5-140 |
| HSUBPD- Subtract Horizontal Double Precision Floating-Point Values | 5-143 |
| HSUBPS- Subtract Horizontal Single Precision Floating-Point Values | 5-145 |
| Hyper-Threading Technology | |
| CPUID flag | 2-53 |

I

| | |
|---|-------|
| IA-32e mode | |
| CPUID flag | 2-42 |
| INSERTPS- Insert Scalar Single Precision Floating-Point Value | 5-149 |

L

| | |
|---|-------|
| L1 Context ID | 2-48 |
| LDDQU- Move Unaligned Integer | 5-153 |
| LDMXCSR instruction | 5-155 |

M

| | |
|--|-------|
| Machine check architecture | |
| CPUID flag | 2-52 |
| description | 2-52 |
| MASKMOVDQU- Store Selected Bytes of Double Quadword with NT Hint | 5-156 |
| MAXPD- Maximum of Packed Double Precision Floating-Point Values | 5-162 |
| MAXPS- Minimum of Packed Single Precision Floating-Point Values | 5-164 |
| MAXSD- Return Maximum Scalar Double-Precision Floating-Point Value | 5-167 |
| MAXSS- Return Maximum Scalar Single-Precision Floating-Point Value | 5-169 |
| MINPD- Minimum of Packed Double Precision Floating-Point Values | 5-171 |
| MINPS- Minimum of Packed Single Precision Floating-Point Values | 5-173 |
| MINSD- Return Minimum Scalar Double-Precision Floating-Point Value | 5-176 |
| MINSS- Return Minimum Scalar Single-Precision Floating-Point Value | 5-178 |
| MMX instructions | |
| CPUID flag for technology | 2-52 |
| Model & family information | 2-57 |
| MONITOR instruction | |
| CPUID flag | 2-48 |
| feature data | 2-57 |
| MOVAPD- Move Aligned Packed Double-Precision Floating-Point Values | 5-180 |
| MOVAPS- Move Aligned Packed Single-Precision Floating-Point Values | 5-183 |
| MOVDDUP- Replicate Double FP Values | 5-191 |
| MOVDQA- Move Aligned Packed Integer Values | 5-193 |
| MOVDQU- Move Unaligned Packed Integer Values | 5-195 |
| MOVD/MOVQ- Move Doubleword and Quadword | 5-186 |
| MOVHLPD - Move Packed Single-Precision Floating-Point Values High to Low | 5-197 |
| MOVHPD- Move High Packed Double-Precision Floating-Point Values | 5-199 |
| MOVHPS- Move High Packed Single-Precision Floating-Point Values | 5-201 |
| MOVLHPS - Move Packed Single-Precision Floating-Point Values Low to High | 5-197 |
| MOVLPD- Move Low Packed Double-Precision Floating-Point Values | 5-205 |
| MOVLPS- Move Low Packed Single-Precision Floating-Point Values | 5-207 |
| MOVMSKPD- Extract Double-Precision Floating-Point Sign mask | 5-209 |

| | |
|--|-------|
| MOVMSKPS- Extract Single-Precision Floating-Point Sign mask | 5-211 |
| MOVNTDQ- Store Packed Integers Using Non-Temporal Hint | 5-213 |
| MOVNTDQA- Load Double Quadword Non-Temporal Aligned Hint. | 5-215 |
| MOVNTPD- Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint | 5-217 |
| MOVNTPS- Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint5- | 219 |
| MOVQ- Move Quadword. | 5-189 |
| MOVSD- Move or Merge Scalar Double-Precision Floating-Point Value | 5-221 |
| MOVSHDUP- Replicate Single FP Values | 5-224 |
| MOVSLDUP- Replicate Single FP Values | 5-227 |
| MOVSS- Move or Merge Scalar Single-Precision Floating-Point Value | 5-230 |
| MOVUPD- Move Unaligned Packed Double-Precision Floating-Point Values | 5-233 |
| MOVUPS- Move Unaligned Packed Single-Precision Floating-Point Values | 5-236 |
| MPSADBW - Multiple Sum of Absolute Differences | 5-239 |
| MULPD- Multiply Packed Double Precision Floating-Point Values. | 5-244 |
| MULPS- Multiply Packed Single Precision Floating-Point Values | 5-246 |
| MULSD- Multiply Scalar Double-Precision Floating-Point Values | 5-248 |
| MULSS- Multiply Scalar Single-Precision Floating-Point Values | 5-250 |
| MWAIT instruction | |
| CPUID flag | 2-48 |
| feature data | 2-57 |

O

| | |
|--|-------|
| ORPD- Bitwise Logical OR of Packed Double Precision Floating-Point Values. | 5-252 |
| ORPS- Bitwise Logical OR of Packed Single Precision Floating-Point Values. | 5-254 |

P

| | |
|--|-------|
| PABSB/PABSW/PABSD - Packed Absolute Value. | 5-256 |
| PACKSSWB/PACKSSDW- Pack with Signed Saturation. | 5-259 |
| PACKUSWB/PACKUSDW- Pack with Unsigned Saturation. | 5-263 |
| PADDB/PADDW/PADDD/PADDQ- Add Packed Integers | 5-266 |
| PADDSB/PADDSW- Add Packed Signed Integers with Signed Saturation | 5-270 |
| PADDUSB/PADDUSW- Add Packed Unsigned Integers with Unsigned Saturation. | 5-272 |
| PALIGNR - Byte Align. | 5-274 |
| PAND- Logical AND | 5-276 |
| PANDN- Logical AND NOT. | 5-278 |
| PAVGB/PAVGW - Average Packed Integers | 5-280 |
| PBLENDVB - Variable Blend Packed Bytes | 5-282 |
| PBLENDW - Blend Packed Words | 5-285 |
| PCLMULQDQ - Carry-Less Multiplication Quadword | 5-287 |
| PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ- Compare Packed Integers for Equality. | 5-298 |
| PCMPESTRI - Packed Compare Explicit Length Strings, Return Index. | 5-290 |
| PCMPESTRM - Packed Compare Explicit Length Strings, Return Mask. | 5-292 |
| PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ- Compare Packed Integers for Greater Than 5- | 302 |
| PCMPISTRI - Packed Compare Implicit Length Strings, Return Index. | 5-294 |
| PCMPISTRM - Packed Compare Implicit Length Strings, Return Mask. | 5-296 |
| Pending break enable. | 2-53 |
| Performance-monitoring counters | |

| | |
|--|-------|
| CPUID inquiry for | 2-57 |
| PEXTRB/PEXTRW/PEXTRD/PEXTRQ- Extract Integer | 5-327 |
| PHADDSW - Packed Horizontal Add with Saturation | 5-333 |
| PHADDW/PHADDD - Packed Horizontal Add | 5-331 |
| PHMINPOSUW - Horizontal Minimum and Position | 5-335 |
| PHSUBSW - Packed Horizontal Subtract with Saturation | 5-339 |
| PHSUBW/PHSUBD - Packed Horizontal Subtract | 5-337 |
| PINSRB/PINSRW/PINSRD/PINSRQ- Insert Integer | 5-341 |
| PMADDUBSW- Multiply and Add Packed Integers | 5-347 |
| PMADDWD- Multiply and Add Packed Integers | 5-345 |
| PMASB/PMASW/PMASD- Maximum of Packed Signed Integers | 5-349 |
| PMASUB/PMASUW/PMASUD- Maximum of Packed Unsigned Integers | 5-353 |
| PMINSB/PMINSW/PMINSUD- Minimum of Packed Signed Integers | 5-357 |
| PMINUB/PMINUW/PMINUD- Minimum of Packed Unsigned Integers | 5-361 |
| PMOVMASKB- Move Byte Mask | 5-365 |
| PMOVSX - Packed Move with Sign Extend | 5-336 |
| PMOVZX - Packed Move with Zero Extend | 5-372 |
| PMULDQ - Multiply Packed Doubleword Integers | 5-388 |
| PMULHRW - Multiply Packed Unsigned Integers with Round and Shift | 5-379 |
| PMULHUW - Multiply Packed Unsigned Integers and Store High Result | 5-377 |
| PMULHW - Multiply Packed Integers and Store High Result | 5-381 |
| PMULLW/PMULLD - Multiply Packed Integers and Store Low Result | 5-383 |
| PMULUDQ - Multiply Packed Unsigned Doubleword Integers | 5-386 |
| POR - Bitwise Logical Or | 5-390 |
| PSADBW - Compute Sum of Absolute Differences | 5-392 |
| PSHUFB - Packed Shuffle Bytes | 5-394 |
| PSHUFD - Shuffle Packed Doublewords | 5-396 |
| PSHUFHW - Shuffle Packed High Words | 5-398 |
| PSHUFLW - Shuffle Packed Low Words | 5-400 |
| PSIGNB/PSIGNW/PSIGND - Packed SIGN | 5-402 |
| PSLLDQ - Byte Shift Left | 5-406 |
| PSLLW/PSLLD/PSLLQ - Bit Shift Left | 5-410 |
| PSRAW/PSRAD - Bit Shift Arithmetic Right | 5-415 |
| PSRLDQ - Byte Shift Right | 5-408 |
| PSRLW/PSRLD/PSRLQ - Shift Packed Data Right Logical | 5-419 |
| PSUBB/PSUBW/PSUBD/PSUBQ -Packed Integer Subtract | 5-428 |
| PSUBSB/PSUBSW -Subtract Packed Signed Integers with Signed Saturation | 5-432 |
| PSUBUB/PSUBUSW -Subtract Packed Unsigned Integers with Unsigned Saturation | 5-434 |
| PTEST- Packed Bit Test | 5-424 |
| PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ - Unpack High Data | 5-436 |
| PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ - Unpack Low Data | 5-440 |
| PXOR - Exclusive Or | 5-444 |

R

| | |
|---|-------|
| RCPPS- Compute Approximate Reciprocals of Packed Single-Precision Floating-Point Values | 5-446 |
| RCPS - Compute Reciprocal of Scalar Single-Precision Floating-Point Value | 5-449 |
| RDMSR instruction | |
| CPUID flag | 2-51 |
| ROUNDPD- Round Packed Double-Precision Floating-Point Values | 5-456 |
| ROUNDPS- Round Packed Single-Precision Floating-Point Values | 5-460 |

| | |
|---|-------|
| ROUNDSD - Round Scalar Double-Precision Value | 5-463 |
| ROUNDSS - Round Scalar Single-Precision Value | 5-465 |
| RSQRTPS - Compute Approximate Reciprocals of Square Roots of Packed Single-Precision Floating-point Values | 5-451 |
| RSQRTSS - Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value 5-454 | |

S

| | |
|--|-------|
| Self Snoop | 2-53 |
| SHUFPS - Shuffle Packed Double Precision Floating-Point Values | 5-467 |
| SHUFPS - Shuffle Packed Single Precision Floating-Point Values | 5-470 |
| SIMD floating-point exceptions, unmasking, effects of | 5-155 |
| SpeedStep technology | 2-48 |
| SQRTPD- Square Root of Double-Precision Floating-Point Values | 5-473 |
| SQRTPS- Square Root of Single-Precision Floating-Point Values | 5-475 |
| SQRTSD - Compute Square Root of Scalar Double-Precision Floating-Point Value | 5-477 |
| SQRTSS - Compute Square Root of Scalar Single-Precision Floating-Point Value | 5-479 |
| SSE extensions | |
| CPUID flag | 2-53 |
| SSE2 extensions | |
| CPUID flag | 2-53 |
| SSE3 | |
| CPUID flag | 2-47 |
| SSE3 extensions | |
| CPUID flag | 2-47 |
| SSSE3 extensions | |
| CPUID flag | 2-48 |
| Stepping information | 2-57 |
| STMXCSR instruction | 5-481 |
| STMXCSR—Store MXCSR Register State | 5-473 |
| SUBPD- Subtract Packed Double Precision Floating-Point Values | 5-481 |
| SUBPS- Subtract Packed Single Precision Floating-Point Values | 5-484 |
| SUBSD- Subtract Scalar Double Precision Floating-Point Values | 5-486 |
| SUBSS- Subtract Scalar Single Precision Floating-Point Values | 5-488 |
| SYSENTER instruction | |
| CPUID flag | 2-51 |
| SYSEXIT instruction | |
| CPUID flag | 2-51 |

T

| | |
|--------------------------|------|
| Thermal Monitor | |
| CPUID flag | 2-53 |
| Thermal Monitor 2 | 2-48 |
| CPUID flag | 2-48 |
| Time Stamp Counter | 2-51 |

U

| | |
|---|-------|
| UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS | 5-490 |
|---|-------|

UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS 5-492

UNPCKHPD- Unpack and Interleave High Packed Double-Precision Floating-Point Values . . 5-494

UNPCKHPS- Unpack and Interleave High Packed Single-Precision Floating-Point Values 5-496

UNPCKLPD- Unpack and Interleave Low Packed Double-Precision Floating-Point Values . . 5-499

UNPCKLPS- Unpack and Interleave Low Packed Single-Precision Floating-Point Values 5-501

V

VBROADCAST- Load with Broadcast 5-52

Version information, processor 2-34

VEX 5-2

VEEXTRACTF128- Extract packed floating-point values 5-134

VEX.B 5-2

VEX.L 5-2

VEX.mmmm 5-2

VEX.pp 5-3

VEX.R 5-4

VEX.vvvv 5-2

VEX.W 5-2

VEX.X 5-2

VFMAADDPD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values 6-2

VFMAADDPSS - Fused Multiply-Add of Packed Single-Precision Floating-Point Values 6-5

VFMAADSD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values 6-8

VFMAADSS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 6-10

VFMAADSUBPD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values 6-12

VFMAADSUBPSS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values 6-14

VFMSUBADDPD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values 6-17

VFMSUBADDPSS - Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values 6-19

VFMSUBPD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values . 6-22

VFMSUBPSS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values . 6-24

VFMSUBSD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values . . 6-27

VFMSUBSS - Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values 6-29

VFNMAADDPD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 6-31

VFNMAADDPSS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values 6-33

VFNMAADSD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values 6-36

VFNMAADSS - Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values. 6-38

VFNMSUBPD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values 6-40

VFNMSUBPSS - Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values 6-43

| | |
|---|-------|
| VFNMSUBSD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values | 6-46 |
| VFNMSUBSS - Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values | 6-48 |
| VINSERTF128- Insert packed floating-point values | 5-148 |
| VMASKMOV- Conditional SIMD Packed Loads and Stores | 5-158 |
| VPERM2F128- Permute Floating-Point Values | 5-325 |
| VPERMIL2PD- Permute 2 Source Double-Precision Floating-Point Values | 5-310 |
| VPERMIL2PS- Permute 2 Source Single-Precision Floating-Point Values | 5-319 |
| VPERMILPD- Permute Double-Precision Floating-Point Values | 5-306 |
| VPERMILPS- Permute Single-Precision Floating-Point Values | 5-315 |
| VZEROALL- Zero All YMM registers. | 5-508 |
| VZERoupper- Zero Upper bits of YMM registers | 5-510 |

W

| | |
|---|------|
| WBINVD/INVD bit | 2-37 |
| WRMSR instruction CPUID flag | 2-51 |

X

| | |
|---|---|
| XFEATURE_ENALBED_MASK | 2-10 |
| XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values | 5-504 |
| XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values | 5-506 |
| XRSTOR | 1-2, 2-10, 2-58, 3-67, 5-12 |
| XSAVE | 1-2, 2-10, 2-11, 2-12, 2-16, 2-49, 2-58, 3-67, 5-12 |