

Intel® Architecture Instruction Set Extensions Programming Reference

319433-013A

JULY 2012

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.

A "MISSION CRITICAL APPLICATION" IS ANY APPLICATION IN WHICH FAILURE OF THE INTEL PRODUCT COULD RESULT, DIRECTLY OR INDIRECTLY, IN PERSONAL INJURY OR DEATH. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The Intel® 64 architecture processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel® Hyper-Threading Technology (Intel® HT Technology) is available on select Intel® Core™ processors. Requires an Intel® HT Technology-enabled system. Consult your PC manufacturer. Performance will vary depending on the specific hardware and software used. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading.

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, and virtual machine monitor (VMM). Functionality, performance or other benefits will vary depending on hardware and software configurations. Software applications may not be compatible with all operating systems. Consult your PC manufacturer. For more information, visit http://www.intel.com/go/virtualization.

Intel® 64 architecture requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit http://www.intel.com/info/em64t.

Intel, Pentium, Intel Atom, Intel Xeon, Intel NetBurst, Intel Core, 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 order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Copyright © 1997-2012 Intel Corporation

CONTENTS

PAGE

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 Overview AVX2 1-4
1.5 Functional Overview 1-5
1.5.1 256-bit Floating-Point Arithmetic Processing Enhancements 1-5
1.5.2 256-bit Non-Arithmetic Instruction Enhancements 1-5
1.5.3 Arithmetic Primitives for 128-bit Vector and Scalar processing 1-6
1.5.4 Non-Arithmetic Primitives for 128-bit Vector and Scalar Processing 1-6
1.5.5 AVX2 and 256-bit Vector Integer Processing 1-7
1.6 General Purpose Instruction Set Enhancements 1-8
1.7 Intel® Transactional Synchronization Extensions 1-8

CHAPTER 2
APPLICATION PROGRAMMING MODEL

2.1 Detection of PCLMULQDQ and AES Instructions 2-1
2.2 Detection of AVX and FMA Instructions 2-1
2.2.1 Detection of FMA 2-3
2.2.2 Detection of VEX-Encoded AES and VPCLMULQDQ 2-4
2.2.3 Detection of AVX2 2-6
2.2.4 Detection of VEX-encoded GPR Instructions 2-7
2.3 Fused-Multiply-ADD (FMA) Numeric Behavior 2-7
2.3.1 FMA Instruction Operand Order and Arithmetic Behavior 2-11
2.4 Accessing YMM Registers 2-12
2.5 Memory alignment 2-13
2.6 SIMD floating-point ExCeptions 2-15
2.7 Instruction Exception Specification 2-15
2.7.1 Exceptions Type 1 (Aligned memory reference) 2-21
2.7.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned) 2-22
2.7.3 Exceptions Type 3 (<16 Byte memory argument) 2-23
2.7.4 Exceptions Type 4 (>=16 Byte mem arg no alignment, no floating-point exceptions) 2-24
2.7.5 Exceptions Type 5 (<16 Byte mem arg and no FP exceptions) 2-25
2.7.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues) 2-26
2.7.7 Exceptions Type 7 (No FP exceptions, no memory arg) 2-27
2.7.8 Exceptions Type 8 (AVX and no memory argument) 2-27
2.7.9 Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions) 2-28
2.7.10 Exception Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions) 2-29
2.7.11 Exception Conditions for VEX-Encoded GPR Instructions 2-30
2.8 Programming Considerations with 128-bit SIMD Instructions 2-31

2.8.1 Clearing Upper YMM State Between AVX and Legacy SSE Instructions 2-32
2.8.2 Using AVX 128-bit Instructions Instead of Legacy SSE instructions 2-33
2.8.3 Unaligned Memory Access and Buffer Size Management 2-33
2.9 CPUID Instruction 2-34
CPUID—CPU Identification 2-34

CHAPTER 3
SYSTEM PROGRAMMING MODEL

3.1 YMM State, VEX Prefix and Supported Operating Modes 3-1
3.2 YMM State Management 3-2
3.2.1 Detection of YMM State Support 3-2
3.2.2 Enabling of YMM State 3-2
3.2.3 Enabling of SIMD Floating-Exception Support 3-3
3.2.4 The Layout of XSAVE Area 3-4
3.2.5 XSAVE/XRSTOR Interaction with YMM State and MXCSR 3-5
3.2.6 Processor Extended State Save Optimization and XSAVEOPT 3-7
3.2.6.1 XSAVEOPT Usage Guidelines 3-8
3.3 Reset Behavior 3-9
3.4 Emulation 3-9
3.5 Writing AVX floating-point exception handlers 3-9

CHAPTER 4
INSTRUCTION FORMAT

4.1 Instruction Formats 4-1
4.1.1 VEX and the LOCK prefix 4-2
4.1.2 VEX and the 66H, F2H, and F3H prefixes 4-2
4.1.3 VEX and the REX prefix 4-2
4.1.4 The VEX Prefix 4-2
4.1.4.1 VEX Byte 0, bits[7:0] 4-4
4.1.4.2 VEX Byte 1, bit [7] - 'R' 4-4
4.1.4.3 3-byte VEX byte 1, bit[6] - 'X' 4-4
4.1.4.4 3-byte VEX byte 1, bit[5] - 'B' 4-6
4.1.4.5 3-byte VEX byte 2, bit[7] - 'W' 4-6
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 4-6
4.1.5 Instruction Operand Encoding and VEX.vvvv, ModR/M 4-7
4.1.5.1 3-byte VEX byte 1, bits[4:0] - "m-mmmm" 4-9
4.1.5.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- "L" 4-9
4.1.5.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- "pp" 4-9
4.1.6 The Opcode Byte 4-10
4.1.7 The MODRM, SIB, and Displacement Bytes 4-10
4.1.8 The Third Source Operand (Immediate Byte) 4-10
4.1.9 AVX Instructions and the Upper 128-bits of YMM registers 4-10
4.1.9.1 Vector Length Transition and Programming Considerations 4-10
4.1.10 AVX Instruction Length 4-11
4.2 Vector SIB (VSIB) Memory Addressing 4-11
4.2.1 64-bit Mode VSIB Memory Addressing 4-13
4.3 VEX Encoding Support for GPR Instructions 4-13

CHAPTER 5
INSTRUCTION SET REFERENCE

5.1 Interpreting Instruction Reference Pages 5-1
5.1.1 Instruction Format 5-1
(V)ADDSD ADD Scalar Double — 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 5-5
5.1.4 Operand Encoding column in the Instruction Summary Table 5-5
5.1.5 64/32 bit Mode Support column in the Instruction Summary Table 5-6
5.1.6 CPUID Support column in the Instruction Summary Table 5-6
5.2 Summary of Terms 5-6
5.3 Instruction SET Reference 5-7
MPSADBW — Multiple Sum of Absolute Differences 5-8
PABSB/PABSW/PABSD — Packed Absolute Value 5-17
PACKSSWB/PACKSSDW — Pack with Signed Saturation 5-21
PACKUSDW — Pack with Unsigned Saturation 5-26
PACKUSWB — Pack with Unsigned Saturation 5-30
PADDB/PADDW/PADDD/PADDQ — Add Packed Integers 5-34
PADDSB/PADDSW — Add Packed Signed Integers with Signed Saturation 5-39
PADDUSB/PADDUSW — Add Packed Unsigned Integers with Unsigned Saturation 5-42
PALIGNR — Byte Align 5-45
PAND — Logical AND 5-48
PANDN — Logical AND NOT 5-50
PAVGB/PAVGW — Average Packed Integers 5-52
PBLENDVB — Variable Blend Packed Bytes 5-55
PBLENDW — Blend Packed Words 5-60
PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ — Compare Packed Integers for Equality 5-63
PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ — Compare Packed Integers for Greater Than 5-68
PHADDW/PHADDD — Packed Horizontal Add 5-73
PHADDSW — Packed Horizontal Add with Saturation 5-77
PHSUBW/PHSUBD — Packed Horizontal Subtract 5-80
PHSUBSW — Packed Horizontal Subtract with Saturation 5-84
PMADDUBSW — Multiply and Add Packed Integers 5-87
PMADDWD — Multiply and Add Packed Integers 5-89
PMAXSB/PMAXSW/PMAXSD — Maximum of Packed Signed Integers 5-92
PMAXUB/PMAXUW/PMAXUD — Maximum of Packed Unsigned Integers 5-97
PMINSB/PMINSW/PMINSD — Minimum of Packed Signed Integers 5-102
PMINUB/PMINUW/PMINUD — Minimum of Packed Unsigned Integers 5-107
PMOVMSKB — Move Byte Mask 5-112
PMOVSX — Packed Move with Sign Extend 5-114
PMOVZX — Packed Move with Zero Extend 5-120
PMULDQ — Multiply Packed Doubleword Integers 5-126
PMULHRSW — Multiply Packed Unsigned Integers with Round and Scale 5-129
PMULHUW — Multiply Packed Unsigned Integers and Store High Result 5-133

PMULHW — Multiply Packed Integers and Store High Result....5-136
PMULLW/PMULLD — Multiply Packed Integers and Store Low Result....5-139
PMULUDQ — Multiply Packed Unsigned Doubleword Integers....5-144
POR — Bitwise Logical Or....5-147
PSADBW — Compute Sum of Absolute Differences....5-149
PSHUFB — Packed Shuffle Bytes....5-152
PSHUFD — Shuffle Packed Doublewords....5-155
PSHUFHW — Shuffle Packed High Words....5-158
PSHUFLW — Shuffle Packed Low Words....5-161
PSIGNB/PSIGNW/PSIGND — Packed SIGN....5-164
PSLLDQ — Byte Shift Left....5-170
PSLLW/PSLLD/PSLLQ — Bit Shift Left....5-172
PSRAW/PSRAD — Bit Shift Arithmetic Right....5-179
PSRLDQ — Byte Shift Right....5-184
PSRLW/PSRLD/PSRLQ — Shift Packed Data Right Logical....5-186
PSUBB/PSUBW/PSUBD/PSUBQ — Packed Integer Subtract....5-193
PSUBSB/PSUBSW — Subtract Packed Signed Integers with Signed Saturation....5-199
PSUBUSB/PSUBUSW — Subtract Packed Unsigned Integers with Unsigned Saturation....5-202
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ — Unpack High Data....5-205
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ — Unpack Low Data....5-212
PXOR — Exclusive Or....5-219
MOVNTDQA — Load Double Quadword Non-Temporal Aligned Hint....5-221
VBROADCAST — Broadcast Floating-Point Data....5-224
VBROADCASTF128/I128 — Broadcast 128-Bit Data....5-226
VPBLENDD — Blend Packed Dwords....5-228
VPBROADCAST — Broadcast Integer Data....5-230
VPERMD — Full Doublewords Element Permutation....5-235
VPERMPD — Permute Double-Precision Floating-Point Elements....5-237
VPERMPS — Permute Single-Precision Floating-Point Elements....5-238
VPERMQ — Qwords Element Permutation....5-240
VPERM2I128 — Permute Integer Values....5-241
VEXTRACTI128 — Extract packed Integer Values....5-243
VINSERTI128 — Insert Packed Integer Values....5-245
VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores....5-247
VPSLLVD/VPSLLVQ — Variable Bit Shift Left Logical....5-251
VPSRAVD — Variable Bit Shift Right Arithmetic....5-254
VPSRLVD/VPSRLVQ — Variable Bit Shift Right Logical....5-256
VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices....5-259
VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices....5-265
VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices....5-271
VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices....5-277

CHAPTER 6
INSTRUCTION SET REFERENCE - FMA

6.1 FMA Instructlon SET Reference 6-1
VFMADD132PD/VFMADD213PD/VFMADD231PD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values 6-2
VFMADD132PS/VFMADD213PS/VFMADD231PS - Fused Multiply-Add of Packed Single-Precision Floating-Point Values 6-6
VFMADD132SD/VFMADD213SD/VFMADD231SD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values 6-10
VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values 6-13
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values 6-16
VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values 6-20
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values 6-24
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS - Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values 6-28
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values 6-32
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values 6-36
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values 6-40
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS - Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values 6-43
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values 6-46
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values 6-50
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values 6-54
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS - Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values 6-57
VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values 6-60
VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS - Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values 6-64
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values 6-68
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS - Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values 6-71

CHAPTER 7
INSTRUCTION SET REFERENCE - VEX-ENCODED GPR INSTRUCTIONS

7.1 Instruction Format 7-1
7.2 INSTRUCTION SET REFERENCE 7-1
ANDN — Logical AND NOT 7-2
BEXTR — Bit Field Extract 7-4
BLSI — Extract Lowest Set Isolated Bit 7-6
BLSMSK — Get Mask Up to Lowest Set Bit 7-8
BLSR — Reset Lowest Set Bit 7-10
BZHI — Zero High Bits Starting with Specified Bit Position 7-12
LZCNT— Count the Number of Leading Zero Bits 7-14
MULX — Unsigned Multiply Without Affecting Flags 7-16
PDEP — Parallel Bits Deposit 7-18
PEXT — Parallel Bits Extract 7-20
RORX — Rotate Right Logical Without Affecting Flags 7-22
SARX/SHLX/SHRX — Shift Without Affecting Flags 7-24
TZCNT — Count the Number of Trailing Zero Bits 7-27
INVPCID — Invalidate Processor Context ID 7-29

CHAPTER 8
INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

8.1 Overview 8-1
8.2 Intel® Transactional Synchronization Extensions 8-1
8.2.1 Hardware Lock Elision 8-2
8.2.2 Restricted Transactional Memory 8-4
8.3 Intel® TSX Application Programming Model 8-4
8.3.1 Detection of Transactional Synchronization Support 8-4
8.3.1.1 Detection of HLE Support 8-4
8.3.1.2 Detection of RTM Support 8-5
8.3.1.3 Detection of XTEST Instruction 8-5
8.3.2 Querying Transactional Execution Status 8-5
8.3.3 Requirements for HLE Locks 8-5
8.3.4 Transactional Nesting 8-6
8.3.4.1 HLE Nesting and Elision 8-6
8.3.4.2 RTM Nesting 8-6
8.3.4.3 Nesting HLE and RTM 8-7
8.3.5 RTM Abort Status Definition 8-7
8.3.6 RTM Memory Ordering 8-8
8.3.7 RTM-Enabled Debugger Support 8-8
8.3.8 Programming Considerations 8-8
8.3.8.1 Instruction Based Considerations 8-8
8.3.8.2 Runtime Considerations 8-10
8.4 Instruction Reference 8-11
XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints 8-12
XABORT — Transaction Abort 8-16
XBEGIN — Transaction Begin 8-18
XEND — Transaction End 8-22
XTEST — Test If In Transactional Execution 8-24

CHAPTER 9
ADDITIONAL NEW INSTRUCTIONS

9.1 Detection of New Instructions. . . . 9-1
9.2 Random Number Instructions 9-1
9.2.1 RDRAND 9-1
9.2.2 RDSEED 9-2
9.2.3 RDSEED and VMX interactions 9-3
9.3 Instruction Exception Specification 9-3
9.4 Instruction Format. . . . 9-4
ADCX — Unsigned Integer Addition of Two Operands with Carry Flag (THIS IS AN EXAMPLE). . . . 9-5
9.5 INSTRUCTION SET REFERENCE 9-5
ADCX — Unsigned Integer Addition of Two Operands with Carry Flag 9-6
ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag 9-8
PREFETCHW—Prefetch Data into Caches in Anticipation of a Write 9-10
RDSEED—Read Random SEED 9-12

APPENDIX A
INSTRUCTION SUMMARY

A.1 AVX Instructions A-1
A.2 Promoted Vector Integer Instructions in AVX2 . . . . A-10

APPENDIX B
OPCODE MAP

B.1 Using Opcode Tables. . . . B-1
B.2 Key to Abbreviations B-2
B.2.1 Codes for Addressing Method B-2
B.2.2 Codes for Operand Type B-3
B.2.3 Register Codes B-4
B.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes B-5
B.2.4.1 One-Byte Opcode Instructions. . . . B-5
B.2.4.2 Two-Byte Opcode Instructions B-6
B.2.4.3 Three-Byte Opcode Instructions. . . . B-7
B.2.4.4 VEX Prefix Instructions B-7
B.2.5 Superscripts Utilized in Opcode Tables. . . . B-8
B.3 One, Two, and THREE-Byte Opcode Maps B-9
B.4 Opcode Extensions For One-Byte And Two-byte Opcodes B-20
B.4.1 Opcode Look-up Examples Using Opcode Extensions B-20
B.4.2 Opcode Extension Tables B-21
B.5 Escape Opcode Instructions B-23
B.5.1 Opcode Look-up Examples for Escape Instruction Opcodes. . . . B-23
B.5.2 Escape Opcode Instruction Tables B-23
B.5.2.1 Escape Opcodes with D8 as First Byte B-24
B.5.2.2 Escape Opcodes with D9 as First Byte B-25
B.5.2.3 Escape Opcodes with DA as First Byte B-26
B.5.2.4 Escape Opcodes with DB as First Byte B-27
B.5.2.5 Escape Opcodes with DC as First Byte B-28

B.5.2.6 Escape Opcodes with DD as First Byte B-29
B.5.2.7 Escape Opcodes with DE as First Byte. B-30
B.5.2.8 Escape Opcodes with DF As First Byte B-31

TABLES

PAGE

2-1 Rounding behavior of Zero Result in FMA Operation 2-8
2-2 FMA Numeric Behavior 2-9
2-3 Alignment Faulting Conditions when Memory Access is Not Aligned..... 2-14
2-4 Instructions Requiring Explicitly Aligned Memory 2-14
2-5 Instructions Not Requiring Explicit Memory Alignment 2-15
2-6 Exception class description 2-16
2-7 Instructions in each Exception Class..... 2-17
2-8 #UD Exception and VEX.W=1 Encoding..... 2-19
2-9 #UD Exception and VEX.L Field Encoding..... 2-19
2-10 Type 1 Class Exception Conditions 2-21
2-11 Type 2 Class Exception Conditions 2-22
2-12 Type 3 Class Exception Conditions 2-23
2-13 Type 4 Class Exception Conditions 2-24
2-14 Type 5 Class Exception Conditions 2-25
2-15 Type 6 Class Exception Conditions 2-26
2-16 Type 7 Class Exception Conditions 2-27
2-17 Type 8 Class Exception Conditions 2-27
2-18 Type 11 Class Exception Conditions..... 2-28
2-19 Type 12 Class Exception Conditions..... 2-29
2-20 Exception Groupings for Instructions Listed in Chapter 7 2-30
2-21 Exception Definition for LZCNT and TZCNT..... 2-30
2-22 Exception Definition (VEX-Encoded GPR Instructions) 2-31
2-23 Information Returned by CPUID Instruction..... 2-35
2-24 Highest CPUID Source Operand for Intel 64 and IA-32 Processors 2-45
2-25 Processor Type Field..... 2-46
2-26 Feature Information Returned in the ECX Register..... 2-49
2-27 More on Feature Information Returned in the EDX Register 2-52
2-28 Encoding of Cache and TLB Descriptors 2-54
2-29 Structured Extended Feature Leaf, Function 0, EBX Register..... 2-59
2-30 Processor Brand String Returned with Pentium 4 Processor..... 2-62
2-31 Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings 2-64
3-1 XFEATURE_ENABLED_MASK and Processor State Components..... 3-3
3-2 CR4 bits for AVX New Instructions technology support..... 3-3
3-3 Layout of XSAVE Area For Processor Supporting YMM State 3-4
3-4 XSAVE Header Format 3-4
3-5 XSAVE Save Area Layout for YMM State (Ext_Save_Area_2) 3-5
3-6 XRSTOR Action on MXCSR, XMM Registers, YMM Registers..... 3-5
3-7 Processor Supplied Init Values XRSTOR May Use..... 3-6
3-8 XSAVE Action on MXCSR, XMM, YMM Register..... 3-7
4-1 VEX.vvvv to register name mapping 4-7
4-2 Instructions with a VEX.vvvv destination 4-8
4-3 VEX.m-mmmm interpretation 4-9
4-4 VEX.L interpretation 4-9
4-5 VEX.pp interpretation..... 4-10

4-6 32-Bit VSIB Addressing Forms of the SIB Byte. 4-12
8-1 RTM Abort Status Definition. 8-7
9-1 Exception Definition (ADCX and ADOX Instructions). 9-4
A-1 Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX. A-1
A-2 Promoted Vector Integer SIMD Instructions in AVX2 . . . . . A-10
A-3 VEX-Only SIMD Instructions in AVX and AVX2 . . . . . A-14
A-4 New Primitive in AVX2 Instructions A-15
A-5 FMA Instructions A-18
A-6 VEX-Encoded and Other General-Purpose Instruction Sets A-27
A-7 New Instructions Introduced in Processors Code Named Ivy Bridge A-29
B-1 Superscripts Utilized in Opcode Tables B-8
B-2 One-byte Opcode Map: (00H — F7H) *. B-10
B-3 Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) * B-12
B-4 Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) * B-16
B-5 Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *. B-18
B-6 Opcode Extensions for One- and Two-byte Opcodes by Group Number * B-21
B-7 D8 Opcode Map When ModR/M Byte is Within 00H to BFH *. B-24
B-8 D8 Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-24
B-9 D9 Opcode Map When ModR/M Byte is Within 00H to BFH *. B-25
B-10 D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-25
B-11 DA Opcode Map When ModR/M Byte is Within 00H to BFH *. B-26
B-12 DA Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-26
B-13 DB Opcode Map When ModR/M Byte is Within 00H to BFH *. B-27
B-14 DB Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-27
B-15 DC Opcode Map When ModR/M Byte is Within 00H to BFH * B-28
B-16 DC Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-28
B-17 DD Opcode Map When ModR/M Byte is Within 00H to BFH *. B-29
B-18 DD Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-29
B-19 DE Opcode Map When ModR/M Byte is Within 00H to BFH * B-30
B-20 DE Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-30
B-21 DF Opcode Map When ModR/M Byte is Within 00H to BFH * B-31
B-22 DF Opcode Map When ModR/M Byte is Outside 00H to BFH *. B-31

FIGURES

PAGE

Figure 2-1. General Procedural Flow of Application Detection of AVX 2-2
Figure 2-2. Version Information Returned by CPUID in EAX..... 2-46
Figure 2-3. Feature Information Returned in the ECX Register..... 2-48
Figure 2-4. Feature Information Returned in the EDX Register 2-51
Figure 2-5. Determination of Support for the Processor Brand String 2-61
Figure 2-6. Algorithm for Extracting Maximum Processor Frequency..... 2-63
Figure 4-1. Instruction Encoding Format with VEX Prefix..... 4-2
Figure 4-2. VEX bitfields..... 4-5
Figure 5-1. VMPSADBW Operation 5-10
Figure 5-2. 256-bit VPALIGN Instruction Operation 5-46
Figure 5-3. 256-bit VPHADDD Instruction Operation 5-74
Figure 5-4. 256-bit VPSHUFD Instruction Operation..... 5-156
Figure 5-5. 128-bit PUNPCKLBW Instruction Operation using 64-bit Operands..... 5-213
Figure 5-6. VBROADCASTI128 Operation..... 5-227
Figure 5-7. VPBROADCASTD Operation (VEX.256 encoded version)..... 5-231
Figure 5-8. VPBROADCASTD Operation (128-bit version)..... 5-231
Figure 5-9. VPERM2I128 Operation 5-241
Figure 7-1. PDEP Example 7-18
Figure 7-2. PEXT Example 7-20
Figure 7-3. INVPCID Descriptor..... 7-29
Figure B-1. ModR/M Byte nnn Field (Bits 5, 4, and 3) B-20

CHAPTER 1
INTEL® ADVANCED VECTOR EXTENSIONS

1.1 ABOUT THIS DOCUMENT

This document describes the software programming interfaces of several vector SIMD and general-purpose instruction extensions of the Intel® 64 architecture that will be introduced with Intel 64 processors built on 22nm process technology. The Intel AVX extensions are introduced in the second generation Intel® Core processor family, and details of Intel AVX are covered in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Additionally, details of VCVTPH2PS/VCVTPS2PH, RDRAND, RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE are also covered there.

The instruction set extensions covered in this document are organized in the following chapters:

- 256-bit vector integer instruction extensions, referred to as Intel® AVX2 (also as AVX2), are described in Chapter 5.
- FMA instruction extensions are described in Chapter 6.
- VEX-encoded, general-purpose instruction extensions are described in Chapter 7.
- Intel Transactional Synchronization Extensions are described in Chapter 8.

Chapter 1 provides an overview of these new instruction set extensions (with Intel AVX included for base reference). Chapter 2 describes the common 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, and extensions for vector-index memory addressing and general-purpose register encoding.

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 addresses 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 1.5. General-purpose encryption and AES instructions follow the existing architecture of 128-bit SIMD instruction sets like SSE4 and its predecessors, Section 1.6 provides a short summary.

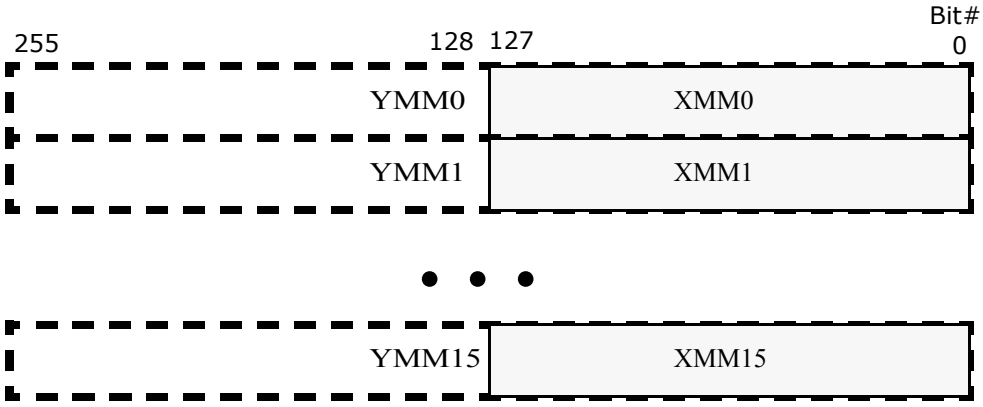
1.3 INTEL® ADVANCED VECTOR EXTENSIONS ARCHITECTURE OVERVIEW

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 generalized three-operand syntax 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.
- FMA extensions and enhanced floating-point compare instructions add support for IEEE-754-2008 standard.

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 a "VEX" prefix). Instruction encoding using the VEX prefix can directly encode a register operand within the VEX prefix. This supports 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., VBLENDVPD, VBLENDVPS, and PBLENDVB).

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.
- 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 OVERVIEW AVX2

AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

1.5 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 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.

1.5.1 256-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 36 256-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.

1.5.2 256-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 39 256-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 18 new data processing instructions that operate on 256-bit vectors. These new primitives cover the following operations:

- Non-unit-stride 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.

1.5.3 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 60 128-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.

1.5.4 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.

1.5.5 AVX2 and 256-bit Vector Integer Processing

AVX2 promotes the vast majority of 128-bit integer SIMD instruction sets to operate with 256-bit wide YMM registers. AVX2 instructions are encoded using the VEX prefix and require the same operating system support as AVX. Generally, most of the promoted 256-bit vector integer instructions follow the 128-bit lane operation, similar to the promoted 256-bit floating-point SIMD instructions in AVX.

Newer functionalities in AVX2 generally fall into the following categories:

- Fetching non-contiguous data elements from memory using vector-index memory addressing. These “gather” instructions introduce a new memory-addressing form, consisting of a base register and multiple indices specified by a vector register (either XMM or YMM). Data elements sizes of 32 and 64-bits are supported, and data types for floating-point and integer elements are also supported.
- Cross-lane functionalities are provided with several new instructions for broadcast and permute operations. Some of the 256-bit vector integer instructions promoted from legacy SSE instruction sets also exhibit cross-lane behavior, e.g. VPMOVZ/VPMOVS family.

- AVX2 complements the AVX instructions that are typed for floating-point operation with a full compliment of equivalent set for operating with 32/64-bit integer data elements.
- Vector shift instructions with per-element shift count. Data elements sizes of 32 and 64-bits are supported.

1.6 GENERAL PURPOSE INSTRUCTION SET ENHANCEMENTS

Enhancements in the general-purpose instruction set consist of several categories:

- A rich collection of instructions to manipulate integer data at bit-granularity. Most of the bit-manipulation instructions employ VEX-prefix encoding to support three-operand syntax with non-destructive source operands. Two of the bit-manipulating instructions (LZCNT, TZCNT) are not encoded using VEX. The VEX-encoded bit-manipulation instructions include: ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, PEXT, PDEP, SARX, SHLX, SHRX, and RORX.
- Enhanced integer multiply instruction (MULX) in conjunctions with some of the bit-manipulation instructions allow software to accelerate calculation of large integer numerics (wider than 128-bits).
- INVPCID instruction targets system software that manages processor context IDs.

Details of enumerating these instruction enhancements are described in detail in Section 2.2.4.

1.7 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

Multithreaded applications take advantage of increasing number of cores to achieve high performance. However, writing multi-threaded applications requires programmers to implement various software mechanisms to handle data sharing among multiple threads. Access to shared data typically requires synchronization mechanisms. These mechanisms ensure multiple threads update shared data by serializing operations on the shared data, often through the use of a critical section protected by a lock.

Since serialization limits concurrency, programmers try to limit synchronization overheads. They do this either through minimizing the use of synchronization or through the use of fine-grain locks; where multiple locks protect different shared data. Unfortunately, this process is difficult and error prone; a missed or incorrect synchronization can cause an application to fail.

Conservatively adding synchronization and using coarser granularity locks, where a few locks each protect many items of shared data, helps avoid correctness problems

but limits performance due to excessive serialization. While programmers must use static information to determine when to serialize, the determination as to whether to actually serialize is best done dynamically.

Intel® Transactional Synchronization Extensions (Intel® TSX) allows the processor to determine dynamically whether threads need to serialize through critical sections, and to perform serialization only when required. This lets processors expose and exploit concurrency hidden in an application due to dynamically unnecessary synchronization.

CHAPTER 2
APPLICATION PROGRAMMING MODEL

The application programming model for AVX2 is the same as Intel AVX and FMA. The VEX-encoded general-purpose instructions generally follows legacy general-purpose instructions. They are summarized as follows:

- Section 2.1 through Section 2.8 apply to AVX2, AVX and FMA. The OS support and detection process is identical for AVX2, AVX, F16C, and FMA
- 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, AESNI, PCLMULQDQ, AVX2, BMI1, BMI2, LZCNT and INVPCID 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/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST, it must check that the processor supports the AES extensions. AES extensions are 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.

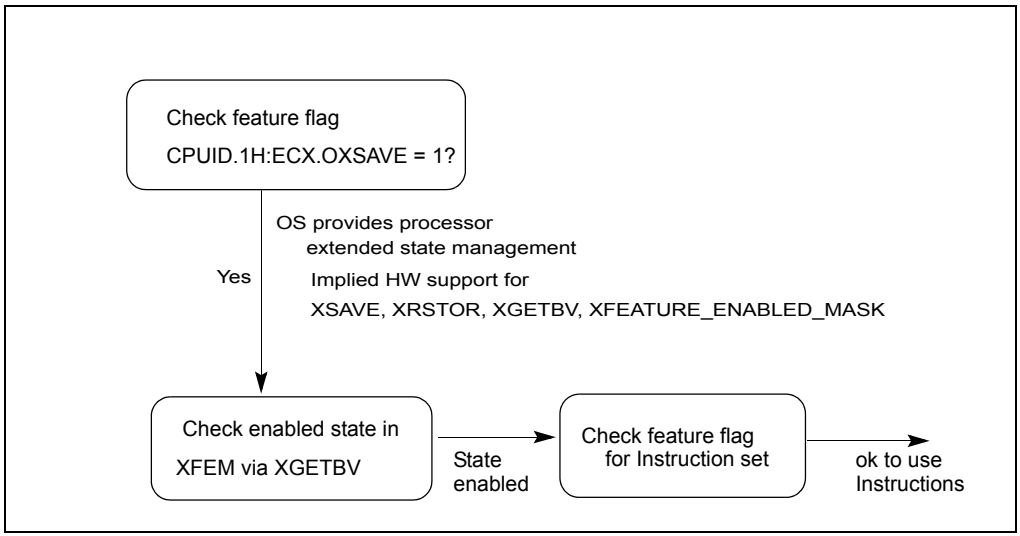


Figure 2-1. General Procedural Flow of Application Detection of AVX

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 use[1])

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_ENALBED_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 systems, 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 256-bit and 128-bit SIMD instructions operating on YMM states.

2.2.2 Detection of VEX-Encoded AES and VPCLMULQDQ

VAESDEC/VAESDECLAST/VAESENC/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.

Similarly, the detection sequence for VPCLMULQDQ must combine checking for CPUID.1:ECX.PCLMULQDQ[bit 1] = 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:
}

INT supports_VPCLMULQDQ()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018000002H
        cmp ecx, 018000002H; check OSXSAVE, AVX and PCLMULQDQ feature flags
         jne not_supported
        ; processor supports AVX and VPCLMULQDQ 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:
}
--------------------------------------------------------------------------------
```

2.2.3 Detection of AVX2

Hardware support for AVX2 is indicated by CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]=1.

Application Software must identify that hardware supports AVX as explained in Section 2.2, after that it must also detect support for AVX2 by checking CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]. The recommended pseudocode sequence for detection of AVX2 is:

```
---------------------------------------------------------------------------------------
INT supports_avx2()
{       ; result in eax
        mov eax, 1
        cpuid
        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 eax, 7
        mov ecx, 0
        cpuid
        and ebx, 20H
        cmp ebx, 20H; check AVX2 feature flags
        jne not_supported
        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:
}
---------------------------------------------------------------------------
```

2.2.4 Detection of VEX-encoded GPR Instructions

VEX-encoded general-purpose instructions do not operate on YMM registers and are similar to legacy general-purpose instructions. Checking for OSXSAVE or YMM support is not required.

There are separate feature flags for the following subsets of instructions that operate on general purpose registers, and the detection requirements for hardware support are:

CPUID.(EAX=07H, ECX=0H):EBX.BMI1[bit 3]: if 1 indicates the processor supports the first group of advanced bit manipulation extensions (ANDN, BEXTR, BLSI, BLSMK, BLSR, TZCNT);

CPUID.(EAX=07H, ECX=0H):EBX.BMI2[bit 8]: if 1 indicates the processor supports the second group of advanced bit manipulation extensions (BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX);

CPUID.(EAX=07H, ECX=0H):EBX.INVPCID[bit 10]: if 1 indicates the processor supports the INVPCID instruction for system software that manages processor context ID.

CPUID.EAX=80000001H:ECX.LZCNT[bit 5]: if 1 indicates the processor supports the LZCNT instruction.

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. Separate FMA instructions are provided to handle different types of arithmetic operations on the three source operands.

FMA instruction syntax is defined using three source operands and the first source operand is updated based on the result of the arithmetic operations of the data elements of 128-bit or 256-bit operands, i.e. The first source operand is also the destination operand.

The 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.

Scalar FMA instructions only perform one arithmetic operation on the low order data element. The content of the rest of the data elements in the lower 128-bits of the destination operand is preserved. the upper 128bits of the destination operand are filled with zero.

An arithmetic FMA operation of the form, r=(x*y)+z, takes two IEEE-754-2008 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 the 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 most significand field equal to 1, and the remaining significand 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
- If the FMA computation represents an invalid operation (e.g. when adding two INF with opposite signs)), the invalid exception is signaled, and the MXCSR.IE flag is set.

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 (multiplicand)	y (multiplier)	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	QNaNIn-definite	QNaNIn-definite	-INF	if x*y and z have the same sign
			QNaNIn-definite	-INF	+INF	QNaNIn-definite	if x*y and z have opposite signs
INF	F, INF	-INF	-INF	QNaNIn-definite	QNaNIn-definite	+INF	if x*y and z have the same sign
			QNaNIn-definite	+INF	-INF	QNaNIn-definite	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	QNaNIn-definite	QNaNIn-definite	QNaNIn-definite	QNaNIn-definite	Signal invalid exception
0	INF	0, F, INF	QNaNIn-definite	QNaNIn-definite	QNaNIn-definite	QNaNIn-definite	Signal invalid exception
F	INF	+INF	+INF	QNaNIn-definite	QNaNIn-definite	-INF	if x*y and z have the same sign
			QNaNIn-definite	-INF	+INF	QNaNIn-definite	if x*y and z have opposite signs

x (multiplicand)	y (multiplier)	z	r=(x*y) +z	r=(x*y) -z	r = -(x*y)+z	r= -(x*y)-z	Comment
F	INF	-INF	-INF	QNaNIn-definite	QNaNIn-definite	+INF	if x*y and z have the same sign
			QNaNIn-definite	+INF	-INF	QNaNIn-definite	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
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	

x (multiplicand)	y (multiplier)	z	r=(x*y) +z	r=(x*y) -z	r = -(x*y)+z	r= -(x*y)-z	Comment
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.

2.3.1 FMA Instruction Operand Order and Arithmetic Behavior

FMA instruction mnemonics are defined explicitly with an ordered three digits, e.g. VFMADD132PD. The value of each digit refers to the ordering of the three source operand as defined by instruction encoding specification:

- '1': The first source operand (also the destination operand) in the syntactical order listed in this specification.
- '2': The second source operand in the syntactical order. This is a YMM/XMM register, encoded using VEX prefix.
- '3': The third source operand in the syntactical order. The first and third operand are encoded following ModR/M encoding rules.

The ordering of each digit within the mnemonic refers to the floating-point data listed on the right-hand side of the arithmetic equation of each FMA operation (see Table 2-2):

- The first position in the three digits of a FMA mnemonic refers to the operand position of the first FP data expressed in the arithmetic equation of FMA operation, the multiplicand.
- The second position in the three digits of a FMA mnemonic refers to the operand position of the second FP data expressed in the arithmetic equation of FMA operation, the multiplier.
- The third position in the three digits of a FMA mnemonic refers to the operand position of the FP data being added/subtracted to the multiplication result.

Note the non-numerical result of an FMA operation does not resemble the mathematically-defined commutative property between the multiplicand and the multiplier values (see Table 2-2). Consequently, software tools (such as an assembler) may support a complementary set of FMA mnemonics for each FMA instruction for ease of programming to take advantage of the mathematical property of commutative multiplications. For example, an assembler may optionally support the complementary mnemonic "VFMADD312PD" in addition to the true mnemonic "VFMADD132PD". The assembler will generate the same instruction opcode sequence corresponding to VFMADD132PD. The processor executes VFMADD132PD and report any NAN conditions based on the definition of VFMADD132PD. Similarly, if the complementary mnemonic VFMADD123PD is supported by an assembler at source level, it must generate the opcode sequence corresponding to VFMADD213PD; the complementary mnemonic VFMADD321PD must produce the opcode sequence defined by VFMADD231PD. In the absence of FMA operations reporting a NAN result, the numerical results of using either mnemonic with an assembler supporting both mnemonics will match the behavior defined in Table 2-2. Support for the complementary FMA mnemonics by software tools is optional.

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 differ 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 CR0.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 && CR0.AM == 1			0	1
Instruction Type	AVX, FMA, AVX2,	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
(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	VMOVNTDQA ymm, m256

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, AVX2, Legacy SSE	16/32 byte explicitly aligned	none
Type 2	AVX, FMA, AVX2, Legacy SSE	16/32 byte not explicitly aligned	yes
Type 3	AVX, FMA, Legacy SSE	< 16 byte	yes
Type 4	AVX, AVX2, Legacy SSE	16/32 byte not explicitly aligned	no
Type 5	AVX, AVX2, Legacy SSE	< 16 byte	no
Type 6	AVX, AVX2 (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, AVX2 Legacy SSE	none	none
Type 8	AVX	none	none
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	yes
Type 12	AVX2	Not explicitly aligned, no AC#	no

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*, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, (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)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADD132SS, VFMADD213SS, VFMADD231SS, VFMSUB132SD, VFMSUB213SD, VFMSUB231SD, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINSD, (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, VBLENDVPD, VBLENDVPS, (V)LDDQU, (V)MASKMOVDQU, (V)PTEST, VTESTPS, VTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (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)PHADDD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW,

Exception Class	Instruction
Type 4 Continued	(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)PMULHRSW, (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, VPBLENDD, VPERMD, VPERMPS, VPERMPD, VPERMQ, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ
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, VLDMXCSR*, VSTMXCSR
Type 6	VEXTRACTF128, VPERMILPD, VPERMILPS, VPERM2F128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**, VMASKMOVPD**, VPMASKMOVD, VPMASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSERTI128, VPERM2I128
Type 7	(V)MOVLHPS, (V)MOVHLPS, (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 11	VCVTPH2PS, VCVTPS2PH
Type 12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

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

(**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no alignment checks are performed.

Table 2-7 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-10 through Table 2-16, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-9 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-8. #UD Exception and VEX.W=1 Encoding

Exception Class	#UD If VEX.W = 1 in all modes	#UD If VEX.W = 1 in non-64-bit modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLENDD, VPERMD, VPERMPS, VPERM2I128, VPSRAVD	
Type 5		VPEXTRQ, VPINSRQ,
Type 6	VEXTRACTF128, VPERMILPD, VPERMILPS, VPERM2F128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	

Table 2-9. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 1		VMOVNTDQA	
Type 2		VDPPD	VDPPD
Type 3			

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 4	VPERMD, VPERMPD, VPERMPS, VPERMQ, VPERM2I128	VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPAND, VPADDUSB/W, VPALIGNR, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPHADDW/D, VPCMPGTB/W/D/Q, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/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, VPUNPCKHQDQ, VPUNPCKHBW/WD/DQ, VPUNPCKLBW/WD/DQ, VPUNPCKLQDQ, VPXOR	VPCMP(E/I)STRI/M, PHMINPOSUW,
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR	same as column 3
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,		
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ	VMOVLHPS, VMOVHLPS,
Type 8			

2.7.1 Exceptions Type 1 (Aligned memory reference)

Table 2-10. 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 2-11. 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 (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
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 2-12. 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 (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		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 2-13. 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 2-14. 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 (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		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 2-15. 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 (F0H)
			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 2-16. 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 2-17. 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 11 (VEX-only, mem arg no AC, floating-point exceptions)

Table 2-18. Type 11 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	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		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

2.7.10 Exception Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions)

Table 2-19. Type 12 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	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	NA	If address size attribute is 16 bit
	X	X	X	X	If ModR/M.mod = '11b'
	X	X	X	X	If ModR/M.rm != '100b'
	X	X	X	X	If any corresponding CPUID feature flag is '0'
	X	X	X	X	If any vector register is used more than once between the destination register, mask register and the index register in VSIB addressing.
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

2.7.11 Exception Conditions for VEX-Encoded GPR Instructions

The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-20 groups instructions listed in Chapter 7 and lists details of the exception conditions for VEX-encoded GRP instructions in Table 2-22 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable. Table 2-21 lists exception conditions for those instructions that support operation on 16-bit operands.

Table 2-20. Exception Groupings for Instructions Listed in Chapter 7

Exception Class	Instruction
See Table 2-21	LZCNT, TZCNT
See Table 2-22	ANDN, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

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

Table 2-21. Exception Definition for LZCNT and TZCNT

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Stack, SS(0)	X	X	X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Table 2-22. Exception Definition (VEX-Encoded GPR Instructions)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'
	X	X			If a VEX prefix is present
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
Stack, SS(0)	X	X	X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference 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 instructions and legacy 128-bit SIMD instructions have internal state to manage the upper and lower halves of the YMM register 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.

In addition to performance considerations, programmers should also be cognizant of the implications of VEX-encoded AVX instructions with the expectations of system software components that manage the processor state components enabled by XCR0. For additional information see Section 4.1.9.1, "Vector Length Transition and Programming Considerations".

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 use 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, PCMPESTRI, PCMPESTRM, PCMPISTRI and 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.

Allocating buffers without regard to the use of the subsequent 16/32 bytes can lead to the rare occurrence of access rights violation as 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.
- 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.

This can be avoided by padding each individual allocation or by padding each area memory is allocated from. As a general rule, the minimal padding size should be the width the largest 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
0F 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 2-23 shows information returned, depending on the initial value loaded into the EAX register. Table 2-24 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. *)
```

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

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

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

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

See also:

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

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

Table 2-23. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 2-24) "Genu" "ntel" "inel"
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 2-26) Feature Information (see Figure 2-4 and Table 2-27) **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 2-28) Cache and TLB Information Cache and TLB Information Cache and TLB Information

Table 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) **NOTES:** Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature. See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on PSN.
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).	
	Deterministic Cache Parameters Leaf	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX. See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 2-58.
	EAX	Bits 4-0: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache 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*, ***, ****
	EBX	Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity*
	ECX	Bits 31-00: S = Number of Sets*

Table 2-23. Information Returned by CPUID Instruction(Continued)

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. Bit 2: Complex cache indexing 0 = Direct mapped cache 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31-03: Reserved = 0 **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.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	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 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWait 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.
Thermal and Power Management Leaf		
06H	EAX	Bits 00: Digital temperature sensor is supported if set Bits 01: Intel Turbo Boost Technology is available 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 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H) Bits 31 - 04: Reserved = 0
	EDX	Reserved = 0
Structured Extended feature Leaf		
07H		**NOTES:** Leaf 07H main leaf (ECX = 0). IF leaf 07H is not supported, EAX=EBX=ECX=EDX=0
	EAX	Bits 31-0: Reports the maximum number sub-leaves that are supported in leaf 07H.

Table 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 02-01: Reserved Bit 03: BMI1 Bit 04: HLE. Bit 05: AVX2 Bit 07: SMEP. Supports Supervisor Mode Execution Protection if 1. Bit 06: Reserved Bit 08: BMI2 Bit 09: ERMS Bit 10: INVPCID Bit 11: RTM Bits 31-12: Reserved
	ECX	Bit 31-0: Reserved
	EDX	Bit 31-0: Reserved.
	Structured Extended Feature Enumeration Sub-leaves (EAX = 07H, ECX = n, n > 1)	
07H		**NOTES:** Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Direct Cache Access Information Leaf	
09H	EAX EBX ECX EDX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H) Reserved Reserved Reserved
	Architectural Performance Monitoring Leaf	

Table 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
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
	EBX	Bit 00: Core cycle event not available if 1 Bit 01: Instruction retired event not available if 1 Bit 02: Reference cycles event not available if 1 Bit 03: Last-level cache reference event not available if 1 Bit 04: Last-level cache misses event not available if 1 Bit 05: Branch instruction retired event not available if 1 Bit 06: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0
	ECX	Reserved = 0 Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1)
	EDX	Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1) Reserved = 0
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** Most of Leaf 0BH 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.
	EAX	Bits 04-00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31-5: Reserved.
	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	Bits 07 - 00: Level number. Same value in ECX input Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 0: x2APIC ID the current logical processor.

Table 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor
	NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.
	** 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. *** 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
Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)	
0DH	**NOTES:** Leaf 0DH main leaf (ECX = 0). EAX: Bits 31-00: Reports the valid bit fields of the lower 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XFEATURE_ENABLED_MASK is reserved. Bit 00: legacy x87 Bit 01: 128-bit SSE Bit 02: 256-bit AVX EBX: Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled. ECX: Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCR0. EDX: Bit 31-0: Reports the valid bit fields of the upper 32 bits of the XFEATURE_ENABLED_MASK register (XCR0). If a bit is 0, the corresponding bit field in XCR0 is reserved
Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	

Table 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EAX EBX ECX EDX	Bit 00: XSAVEOPT is available; Bits 31-1: Reserved Reserved Reserved Reserved
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Each valid sub-leaf index maps to a valid bit in the XCR0 register starting at bit position 2
	EAX	Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	EBX	Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
		*The highest valid sub-leaf index, *n*, is (POPCNT(CPUID.(EAX=0D, ECX=0):EAX) + POPCNT(CPUID.(EAX=0D, ECX=0):EDX) - 1)
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 2-24).
	EBX ECX EDX	Reserved Reserved Reserved
80000001H	EAX EBX ECX	Extended Processor Signature and Feature Bits. Reserved Bit 0: LAHF/SAHF available in 64-bit mode Bits 4-1: Reserved Bit 5: LZCNT available Bits 31-6 Reserved

Table 2-23. Information Returned by CPUID Instruction(Continued)

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 2-23. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
80000007H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	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	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
		NOTES: * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 2-24) 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 2-24) and is processor specific.

Table 2-24. 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 10 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 2-25 for available processor type values. Stepping IDs are provided as needed.

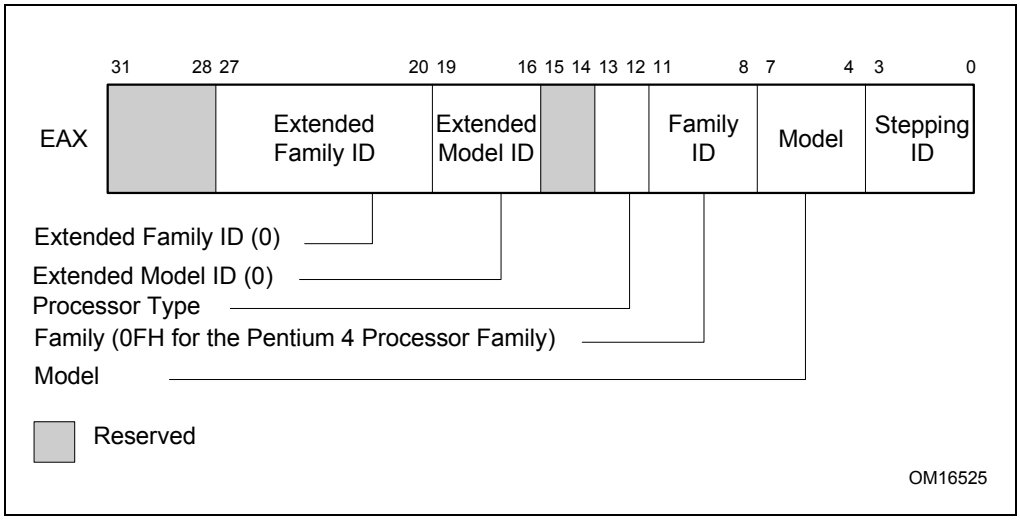


Figure 2-2. Version Information Returned by CPUID in EAX

Table 2-25. Processor Type Field

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

NOTE

See "Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual,*

Volume 3A, and Chapter 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 2-26 show encodings for ECX.
- Figure 2-4 and Table 2-27 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.

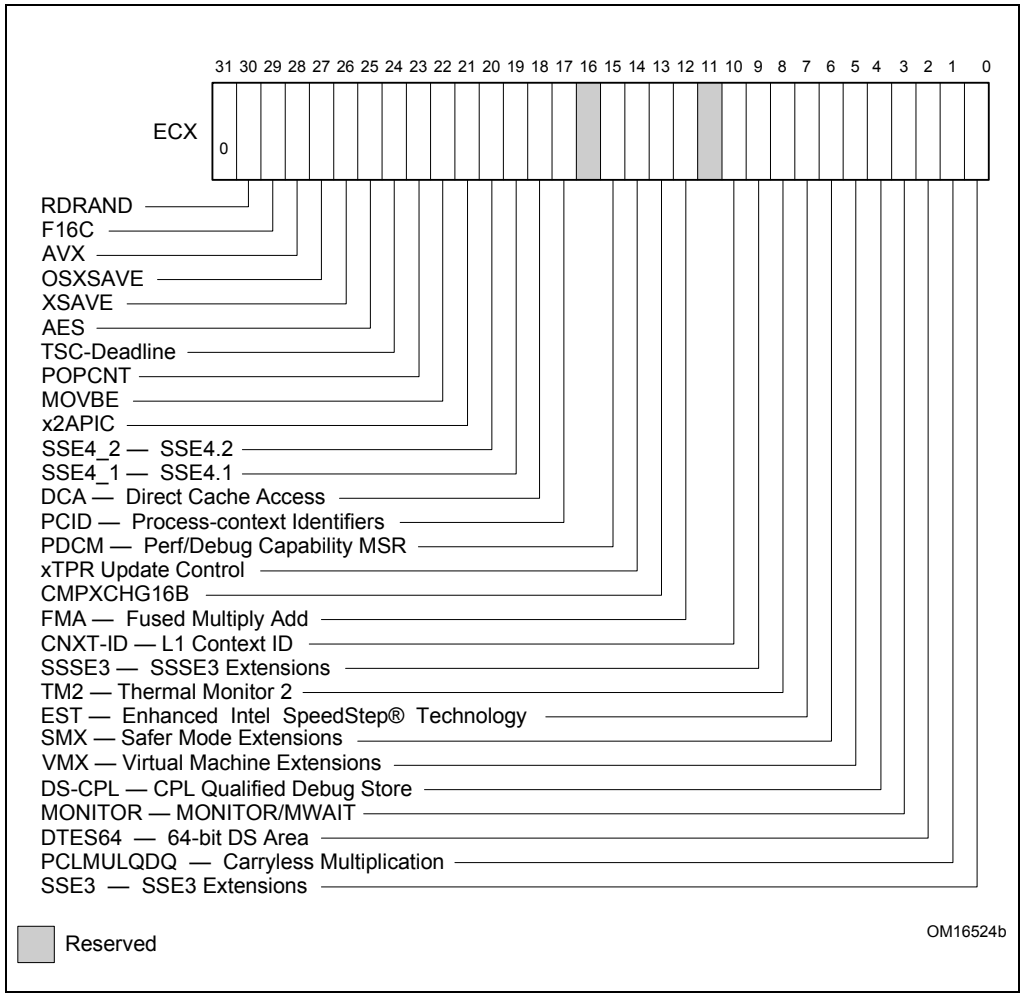


Figure 2-3. Feature Information Returned in the ECX Register

Table 2-26. 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	DTES64	**64-bit DS Area**. A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT**. A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store**. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 5, "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.
14	xTPR Update Control	**xTPR Update Control**. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
15	PDCM	**Perfmon and Debug Capability**: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved

Table 2-26. Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AES	A value of 1 indicates that the processor supports the AES instruction.
26	XSAVE	A value of 1 indicates that the processor supports the XFEATURE_ENABLED_MASK register and XSAVE/XRSTOR/XSETBV/XGETBV instructions to manage processor extended states.
27	OSXSAVE	A value of 1 indicates that the OS has enabled support for using XGETBV/XSETBV instructions to query processor extended states.
28	AVX	A value of 1 indicates that processor supports AVX instructions operating on 256-bit YMM state, and three-operand encoding of 256-bit and 128-bit SIMD instructions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always return 0

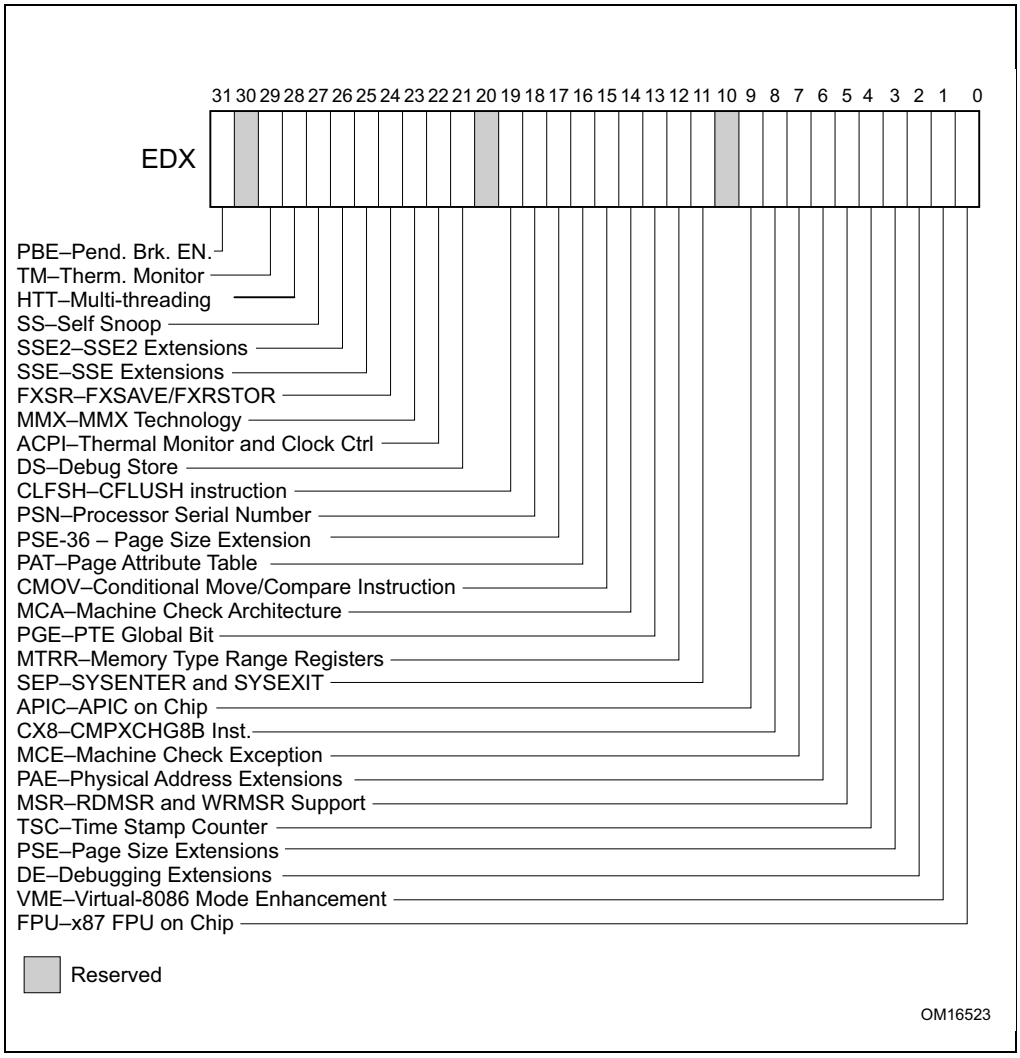


Figure 2-4. Feature Information Returned in the EDX Register

Table 2-27. 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 2-27. More on Feature Information Returned in the EDX Register(Continued)

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 17, "Debugging, Branch Profiles and Time-Stamp Counter," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
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 2-27. More on Feature Information Returned in the EDX Register(Continued)

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 2-28 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 2-28. 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 2-28. Encoding of Cache and TLB Descriptors (Continued)

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 2-28. Encoding of Cache and TLB Descriptors (Continued)

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-µop, 8-way set associative
71H	Trace cache: 16 K-µop, 8-way set associative
72H	Trace cache: 32 K-µop, 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 2-28. Encoding of Cache and TLB Descriptors (Continued)

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-μop, 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 2-23.

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 8, "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 2-23.

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 2-23.

INPUT EAX = 7: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 7 and ECX = 0, the processor returns information about the maximum number of sub-leaves that contain extended feature flags. See Table 2-23.

When CPUID executes with EAX set to 7 and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=07H, ECX= 0H).EAX, the processor returns information about extended feature flags. See Table 2-23. In sub-leaf 0, only EAX has the number of sub-leaves. In sub-leaf 0, EBX, ECX & EDX all contain extended feature flags.

Table 2-29. Structured Extended Feature Leaf, Function 0, EBX Register

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

INPUT EAX = 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 2-23.

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 2-23) is greater than Pn 0. See Table 2-23.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 17, "Debugging, Branch Profiles and Time-Stamp Counter," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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 >= 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 2-23.

When CPUID executes with EAX set to 13 and ECX = n (n > 1and 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 2-23.

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.

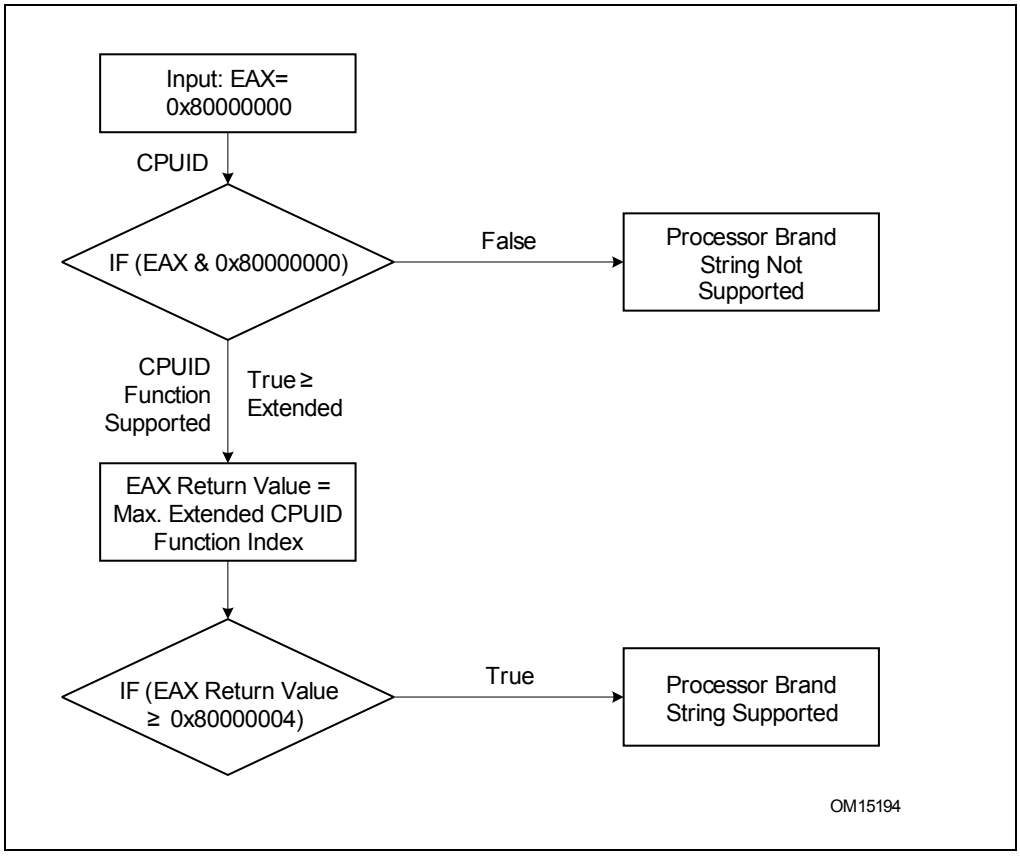


Figure 2-5. Determination of Support for the Processor Brand String

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 2-30 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 2-30. Processor Brand String Returned with Pentium 4 Processor

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

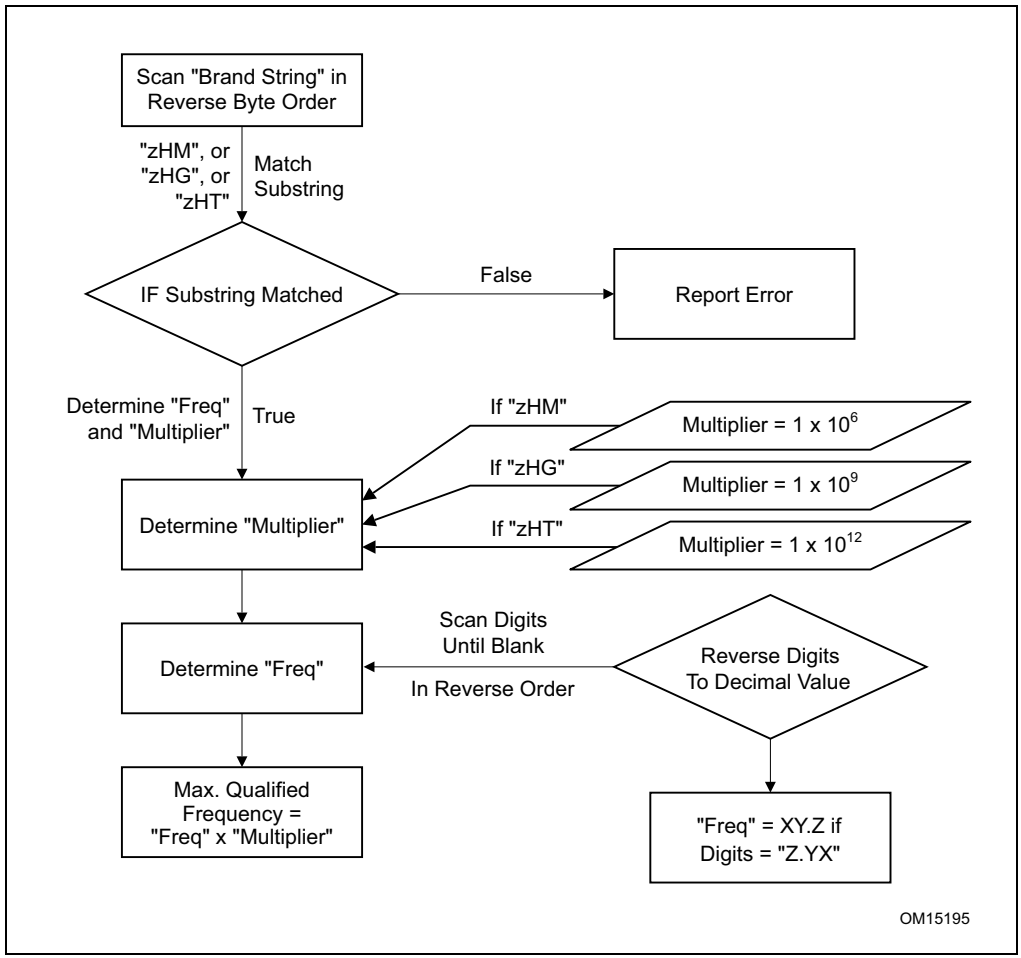


Figure 2-6. Algorithm for Extracting Maximum Processor Frequency

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 2-31 shows brand indices that have identification strings associated with them.

Table 2-31. 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 2-23. *)
    EBX ← Deterministic Cache Parameters Leaf;
    ECX ← Deterministic Cache Parameters Leaf;
    EDX ← Deterministic Cache Parameters Leaf;
BREAK;
EAX = 5H:
    EAX ← MONITOR/MWAIT Leaf; (* See Table 2-23. *)
    EBX ← MONITOR/MWAIT Leaf;
    ECX ← MONITOR/MWAIT Leaf;
    EDX ← MONITOR/MWAIT Leaf;
BREAK;
EAX = 6H:
    EAX ← Thermal and Power Management Leaf; (* See Table 2-23. *)
    EBX ← Thermal and Power Management Leaf;
    ECX ← Thermal and Power Management Leaf;
    EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H :
    EAX ← Structured Extended Feature Leaf; (* See Table 2-23. *);
    EBX ← Structured Extended Feature Leaf;
    ECX ← Structured Extended Feature Leaf;
    EDX ← Structured Extended Feature Leaf;
BREAK;
EAX = 8H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 2-23. *)
    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 2-23. *)
    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 2-23. *)
        EBX ← Extended Topology Enumeration Leaf;
        ECX ← Extended Topology Enumeration Leaf;
        EDX ← Extended Topology Enumeration Leaf;
    BREAK;
    EAX = CH:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = DH:
        EAX ← Processor Extended State Enumeration Leaf; (* See Table 2-23. *)
        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 2-23.*);
        EDX ← Extended Feature Bits (* See Table 2-23. *);
    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, F16C, AVX2 and FMA. The AES extensions 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 *IA-32 Intel Architecture Software Developer's Manual, Volumes 3A*).

The AVX, F16C, AVX2 and FMA extensions operate on 256-bit YMM registers, and require operating system to supports processor extended state management using XSAVE/XRSTOR instructions. VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST/VPCLMULQDQ 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 *IA-32 Intel Architecture 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, F16C, AVX2 and FMA instructions operates on YMM states and requires 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. Not all instructions that require VEX prefix encoding need YMM or XMM registers as operands.

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 XSAVE/XRSTOR. This bit must be 1 if CPUID.01H:ECX.XSAVE[26] = 1.
1 - SSE	If set, the processor supports SSE state (XMM and MXCSR) management via XSAVE/XRSTOR. 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 XSAVE. 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
CR4.OSXSAVE[bit 18]	If set, the OS supports use of XSETBV/XGETBV instruction to access the XFEATURE_ENABLED_MASK register, XSAVE/XRSTOR to manage processor extended state. Must be set to '1' to enable AVX and FMA.
CR4.OSXMMEXCPT[bit 10]	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.
CR4.OSFXSR[bit 9]	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 buffer 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.

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. MXCSR save/restore will not be bound to any future states.

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

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	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_ENABLED_MASK		MXCSR	YMM_H Registers	XMM Registers
Bit 2	Bit 1	Bit 2	Bit 1			
0	0	X	X	None	None	None
0	1	X	1	Store	None	Store
0	1	X	0	None	None	None
1	0	0	X	None	None	None
1	0	1	1	Store	Store	None
1	1	0	0	None	None	None
1	1	0	1	Store	None	Store
1	1	1	1	Store	Store	Store

3.2.6 Processor Extended State Save Optimization and XSAVEOPT

The XSAVEOPT instruction paired with XRSTOR is designed to provide a high performance method for system software to perform state save and restore.

A processor may indicate its support for the XSAVEOPT instruction if CPUID.(EAX=0DH, ECX=1):EAX.XSAVEOPT[Bit 0] = 1. The functionality of XSAVEOPT is similar to XSAVE. Software can use XSAVEOPT/XRSTOR in a pair-wise manner similar to XSAVE/XRSTOR to save and restore processor extended states.

The syntax and operands for XSAVEOPT instructions are identical to XSAVE, i.e. the mask operand in EDX:EAX specifies the subset of enabled features to be saved.

Note that software using XSAVEOPT must observe the same restrictions as XSAVE while allocating a new save area. i.e., the header area must be initialized to zeroes. The first 64-bits in the save image header starting at offset 512 are referred to as XHEADER.BV. However, the instruction differs from XSAVE in several important aspects:

1. If a component state in the processor specified by the save mask corresponds to an INIT state, the instruction may clear the corresponding bit in XHEADER.BV, but may not write out the state (unlike the XSAVE instruction, which always writes out the state).
2. If the processor determines that the component state specified by the save mask hasn't been modified since the last XRSTOR, the instruction may not write out the state to the save area.
3. A implication of this optimization is that software which needs to examine the saved image must first check the XHEADER.BV to see if any bits are clear. If the

header bit is clear, it means that the state is INIT and the saved memory image may not correspond to the actual processor state.

4. The performance of XSAVEOPT will always be better than or at least equal to that of XSAVE.

3.2.6.1 XSAVEOPT Usage Guidelines

When using the XSAVEOPT facility, software must be aware of the following guidelines:

1. The processor uses a tracking mechanism to determine which state components will be written to memory by the XSAVEOPT instruction. The mechanism includes three sub-conditions that are recorded internally each time XRSTOR is executed and evaluated on the invocation of the next XSAVEOPT. If a change is detected in any one of these sub-conditions, XSAVEOPT will behave exactly as XSAVE. The three sub-conditions are:
   — current CPL of the logical processor
   — indication whether or not the logical processor is in VMX non-root operation
   — linear address of the XSAVE/XRSTOR area
2. Upon allocation of a new XSAVE/XRSTOR area and before an XSAVE or XSAVEOPT instruction is used, the save area header (HEADER.XSTATE) must be initialized to zeroes for proper operation.
3. XSAVEOPT is designed primarily for use in context switch operations. The values stored by the XSAVEOPT instruction depend on the values previously stored in a given XSAVE area.
4. Manual modifications to the XSAVE area between an XRSTOR instruction and the matching XSAVEOPT may result in data corruption.
5. For optimization to be performed properly, the XRSTOR XSAVEOPT pair must use the same segment when referencing the XSAVE area and the base of that segment must be unchanged between the two operations.
6. Software should avoid executing XSAVEOPT into a buffer from which it hadn't previously executed a XRSTOR. For newly allocated buffers, software can execute XRSTOR with the linear address of the buffer and a restore mask of EDX:EAX = 0. Executing XRSTOR(0:0) doesn't restore any state, but ensures expected operation of the XSAVEOPT instruction.
7. The XSAVE area can be moved or even paged, but the contents at the linear address of the save area at an XSAVEOPT must be the same as that when the previous XRSTOR was performed.

A destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception being generated. In 64-bit mode the upper 32 bits of RDX and RAX are ignored.

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.

CHAPTER 4
INSTRUCTION FORMAT

AVX F16C, AVX2 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 uses 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.

VEX-prefix encoding enables a subset of AVX2 instructions to support "vector SIM" form of memory addressing. This is described in Section 4.2.

VEX-prefix encoding also enables some general purpose instructions to support three-operand syntax. This is described in Section 4.3.

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 VBLENDVPD 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 (0FH), 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.

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.

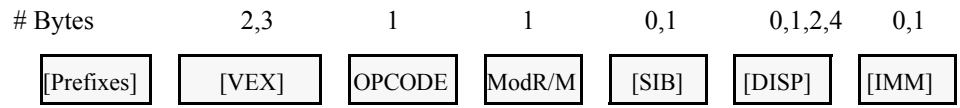


Figure 4-1. Instruction Encoding Format with VEX Prefix

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 proceeding 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.
- 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.

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. VEX-encoded 128-bit vector integer instructions are supported in AVX. 256-bit vector integer instructions are supported in AVX2 but not AVX. Table A-1 of Appendix A lists promoted, VEX-128 encoded vector integer instructions in AVX. Table A-2 lists 128-bit and 256-bit, promoted VEX-encoded vector integer instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix (See Appendix A, Table A-3, Table A-4, Table A-5).

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

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

4.1.4.1 VEX Byte 0, bits[7:0]

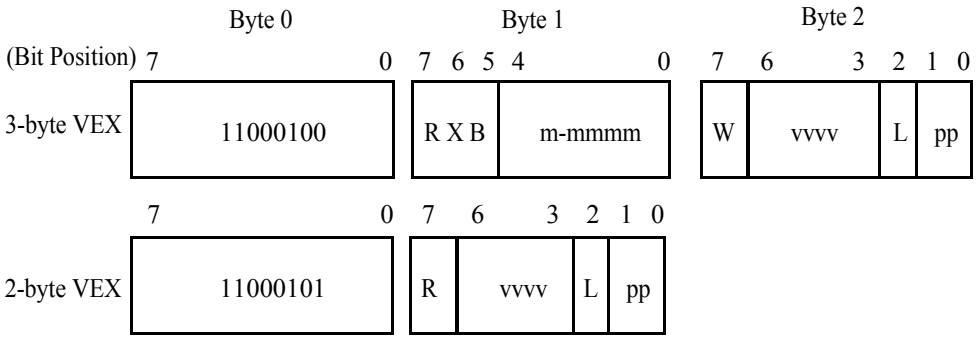
VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

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.



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 opcode extension, or ignored, depending on the opcode byte)

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

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 (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), 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 (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

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.

- 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 three 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.
- For the gather family of instructions in AVX2, ModR/M.r/m support vector SIB memory addressing (see Section 4.2).

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 the following situation:

- Imm8[7:4] encodes the third source register operand.

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 4-3. 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 4-4. 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 4-5. 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 instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

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.

4.1.9.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255.

(This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

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.

4.2 VECTOR SIB (VSIB) MEMORY ADDRESSING

In AVX2, an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.

- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 4-6 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8L-R15L applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 4-6). In 32-bit mode, R8L-R15L does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e. XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

Table 4-6. 32-Bit VSIB Addressing Forms of the SIB Byte

r32 (In decimal) Base = (In binary) Base =				EAX/ R8L 0 000	ECX/ R9L 1 001	EDX/ R10L 2 010	EBX/ R11L 3 011	ESP/ R12L 4 100	EBP/ R13L[1] 5 101	ESI/ R14L 6 110	EDI/ R15L 7 111
Scaled Index		**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
VR0/VR8	*1	00	000	00	01	02	03	04	05	06	07
VR1/VR9			001	08	09	0A	0B	0C	0D	0E	0F
VR2/VR10			010	10	11	12	13	14	15	16	17
VR3/VR11			011	18	19	1A	1B	1C	1D	1E	1F
VR4/VR12			100	20	21	22	23	24	25	26	27
VR5/VR13			101	28	29	2A	2B	2C	2D	2E	2F
VR6/VR14			110	30	31	32	33	34	35	36	37
VR7/VR15			111	38	39	3A	3B	3C	3D	3E	3F
VR0/VR8	*2	01	000	40	41	42	43	44	45	46	47
VR1/VR9			001	48	49	4A	4B	4C	4D	4E	4F
VR2/VR10			010	50	51	52	53	54	55	56	57
VR3/VR11			011	58	59	5A	5B	5C	5D	5E	5F
VR4/VR12			100	60	61	62	63	64	65	66	67
VR5/VR13			101	68	69	6A	6B	6C	6D	6E	6F
VR6/VR14			110	70	71	72	73	74	75	76	77
VR7/VR15			111	78	79	7A	7B	7C	7D	7E	7F
VR0/VR8	*4	10	000	80	81	82	83	84	85	86	87
VR1/VR9			001	88	89	8A	8B	8C	8D	8E	8F
VR2/VR10			010	90	91	92	93	94	95	96	97
VR3/VR11			011	98	89	9A	9B	9C	9D	9E	9F
VR4/VR12			100	A0	A1	A2	A3	A4	A5	A6	A7
VR5/VR13			101	A8	A9	AA	AB	AC	AD	AE	AF
VR6/VR14			110	B0	B1	B2	B3	B4	B5	B6	B7
VR7/VR15			111	B8	B9	BA	BB	BC	BD	BE	BF

Table 4-6. 32-Bit VSIB Addressing Forms of the SIB Byte

VR0/VR8	*8	11	000	C0	C1	C2	C3	C4	C5	C6	C7
VR1/VR9			001	C8	C9	CA	CB	CC	CD	CE	CF
VR2/VR10			010	D0	D1	D2	D3	D4	D5	D6	D7
VR3/VR11			011	D8	D9	DA	DB	DC	DD	DE	DF
VR4/VR12			100	E0	E1	E2	E3	E4	E5	E6	E7
VR5/VR13			101	E8	E9	EA	EB	EC	ED	EE	EF
VR6/VR14			110	F0	F1	F2	F3	F4	F5	F6	F7
VR7/VR15			111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13]+ disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

MOD	Effective Address
00b	[Scaled Vector Register] + Disp32
01b	[Scaled Vector Register] + Disp8 + [EBP/R13]
10b	[Scaled Vector Register] + Disp32 + [EBP/R13]

4.2.1 64-bit Mode VSIB Memory Addressing

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 4-6 base register should be interpreted as the full 64-bit of each register.

4.3 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape 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 or memory addressing.
- 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.

- VEX-encoded GPR instructions are encoded with VEX.L=0.

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

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

VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

§

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:

(V)ADDSD ADD Scalar Double — Precision Floating-Point Values (THIS IS AN EXAMPLE)

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1
VEX.NDS.128.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1

Instruction Operand Encoding

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

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,NDD,DDS].[128,256,LZ,LIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1, WIG] opcode [/r] [/ib,/is4]

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

  - VEX.NDS: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
  - VEX.NDD: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
  - VEX.DDS: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.
  - If none of NDS, NDD, and DDS is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.

— **128,256,LZ,LIG:** VEX.L field can be 0 (denoted by VEX.128 or VEX.LZ) 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: Two 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).
  - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. 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.
  - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.

— **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) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 4.1.4 on the subfield definitions within VEX.
- **WIG:** If WIG is present, the instruction may be encoded using the C5H form (if VEX.mmmmm is not required); or when using the C4H form of VEX prefix, VEX.W value is ignored.

- **opcode:** Instruction opcode.
- **/r** — Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **ib**: A 1-byte immediate operand to the instruction that follows the opcode, ModR/M bytes or scale/indexing bytes.
- **/is4:** An 8-bit immediate byte is present specifying a source register in imm[7:4] and containing an instruction-specific payload in imm[3:0].
- In general, the encoding of the VEX.R, VEX.X, and VEX.B fields are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, and 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.
- **vm32x,vm32y** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value. The vector index register is an XMM register if expressed as **vm32x**. The vector index register is a YMM register if expressed as **vm32y**.
- **vm64x,vm64y** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual element of 64-bit index value. The vector index register is an XMM register if expressed as **vm64x**. The vector index register is a YMM register if expressed as **vm64y**.
- **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 Operand Encoding column in the Instruction Summary Table

The “operand encoding” column is abbreviated as Op/En in the Instruction Summary table heading. Each entry corresponds to a specific instruction syntax in the immediate column to its left and points to a corresponding row in a separate instruction operand encoding table immediately following the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

5.1.5 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.

5.1.6 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, CPUID.(EAX=7,ECX=0).EBX, CPUID.80000001H.ECX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AVX/FP16/RDRAND/AVX2/BMI1/BMI2/LZCNT support) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

5.2 SUMMARY OF TERMS

- **"Legacy SSE":** Refers to SSE, SSE2, SSE3, SSSE3, SSE4, and any future instruction sets referencing XMM registers and encoded without a VEX 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
- **VLMAX:** the maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's prefix but is instead determined by the current value of XCR0. For existing processors, VLMAX is 256 whenever XFEATURE_ENABLED_MASK.YMM[bit 2] is 1. Future processors may defined new bits in XFEATURE_ENABLED_MASK whose setting may imply other values for VLMAX.

VLMAX Definition

XCR0 Component	VLMAX
XCR0.YMM	256

5.3 INSTRUCTION SET REFERENCE

<AVX2 instructions are listed below>

MPSADBW — Multiple Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F3A 42 /r ib MPSADBW xmm1, xmm2/m128, imm8	A	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.WIG 42 /r ib VMPSADBW xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and xmm3/m128 and writes the results in xmm1. Starting offsets within xmm2 and xmm3/m128 are determined by imm8.
VEX.NDS.256.66.0F3A.WIG 42 /r ib VMPSADBW ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and ymm3/m128 and writes the results in ymm1. Starting offsets within ymm2 and xmm3/m128 are determined by imm8.

Instruction Operand Encoding

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

Description

(V)MPSADBW sums the absolute difference of 4 unsigned bytes (block_2) in the second source operand with sequential groups of 4 unsigned bytes (block_1) in the first source operand. The immediate byte provides bit fields that specify the initial offset of block_1 within the first source operand, and the offset of block_2 within the second source operand. The offset granularity in both source operands are 32 bits. The sum-absolute-difference (SAD) operation is repeated 8 times for (V)MPSADW

between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations is written to the respective word in the destination operand.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

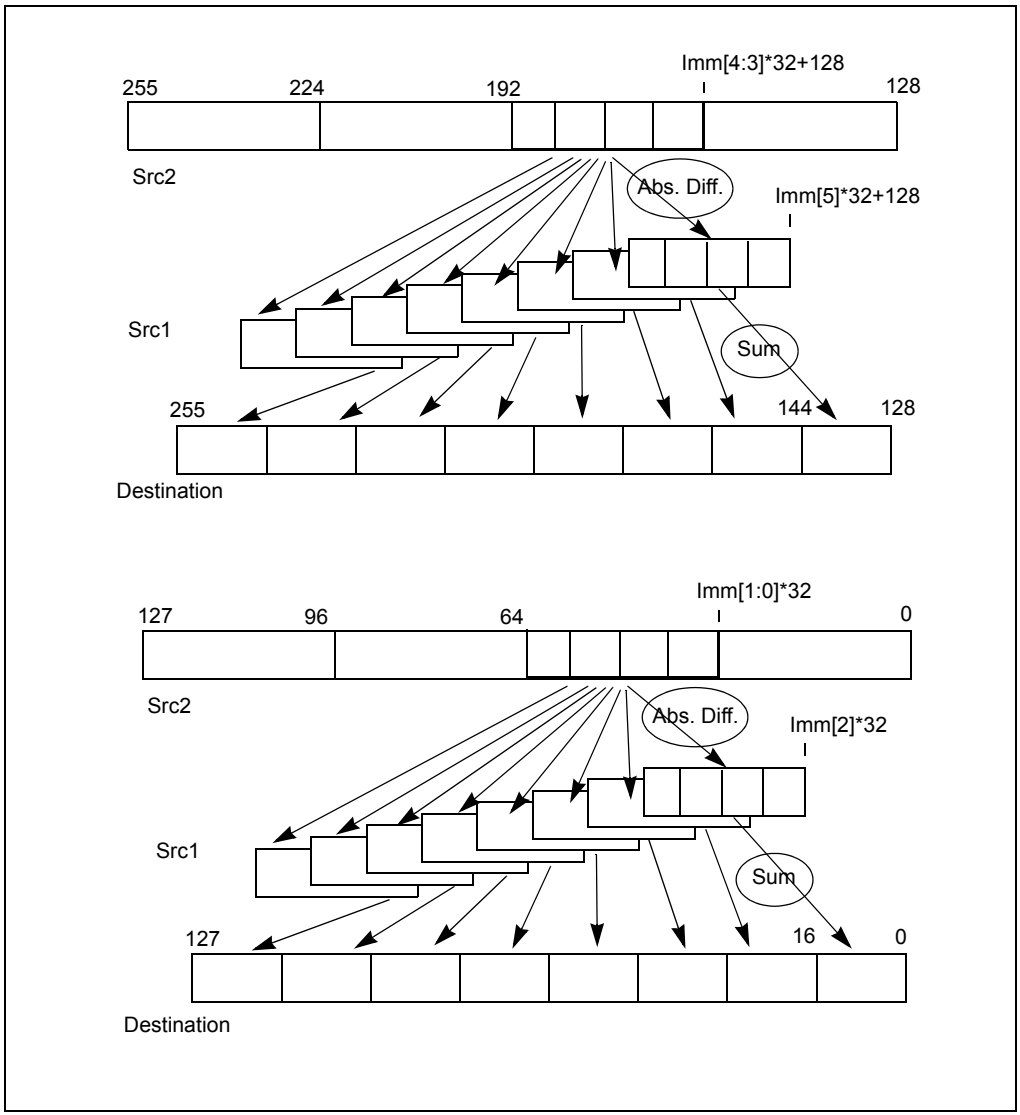


Figure 5-1. VMPSADBW Operation

Operation

VMPSADBW (VEX.256 encoded version)

SRC2_OFFSET ← imm8[1:0]*32
SRC1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]

```
SRC1_BYTE1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTE2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTE3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTE4 ←SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTE5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYTE6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYTE7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYTE8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYTE9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYTE10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

SRC2_BYTE0 ←SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYTE1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYTE2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYTE3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
```

```
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

SRC2_OFFSET ← imm8[4:3]*32 + 128
SRC1_OFFSET ← imm8[5]*32 + 128
SRC1_BYTE0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTE1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTE2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTE3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTE4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTE5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYTE6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYTE7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYTE8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYTE9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYTE10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

SRC2_BYTE0 ←SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYTE1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYTE2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYTE3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[143:128] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ←ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
```

```
DEST[159:144] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[175:160] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ←ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[191:176] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[207:192] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[223:208] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[239:224] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[255:240] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

VMPSADBW (VEX.128 encoded version)

```
SRC2_OFFSET ← imm8[1:0]*32
SRC1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTE1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTE2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTE3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTE4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTE5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYTE6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
```

```
SRC1_BYTE7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYTE8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYTE9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYTE10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

SRC2_BYTE0 ←SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYTE1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYTE2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYTE3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
```

```
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

DEST[VLMAX:128] ← 0
```

MPSADBW (128-bit Legacy SSE version)

```
SRC_OFFSET ← imm8[1:0]*32
DEST_OFFSET ← imm8[2]*32
DEST_BYTE0 ← DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 ← DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 ← DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 ← DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 ← DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 ← DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 ← DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 ← DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 ← DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 ← DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 ← DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 ←SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYTE1 ← SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYTE2 ←SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYTE3 ← SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMP0 ← ABS(DEST_BYTE0 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE1 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE2 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE1 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE2 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE3 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE2 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE3 - SRC_BYTE1)
```

```
TEMP2 ← ABS(DEST_BYTE4 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE3 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE4 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE5 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE4 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE5 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE6 - SRC_BYTE2)
TEMP3 ←ABS(DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
TEMP0 ← ABS(DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TE
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)MPSADBW: __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);

VMPSADBW: __m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PABSB/PABSW/PABSD — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 1C /r PABSB xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSB xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.256.66.0F38.WIG 1C /r VPABSB ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 1D /r VPABSW ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.
VEX.256.66.0F38.WIG 1E /r VPABSD ymm1, ymm2/m256	A	V/V	AVX2	Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.

Instruction Operand Encoding

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

Description

(V)PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). (V)PABSB operates on signed bytes, (V)PABSW operates on 16-bit words, and (V)PABSD operates on signed 32-bit integers. The source operand can be an XMM register or a YMM register or a 128-bit memory location or 256-bit memory location. The destination operand can be an XMM or a YMM register. Both operands can be MMX register or XMM registers.

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

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

128-bit Legacy SSE version: The source operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (255:128) of the corresponding YMM register destination are unmodified.

Operation

PABSB with 128 bit operands:

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

VPABSB with 128 bit operands:

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

VPABSB with 256 bit operands:

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

PABSW with 128 bit operands:

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

VPABSW with 128 bit operands:

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

VPABSW with 256 bit operands:

Unsigned DEST[15:0]← ABS(SRC[15:0])
Repeat operation for 2nd through 15th 16-bit words
Unsigned DEST[255:240] ← ABS(SRC[255:240])

PABSD with 128 bit operands:

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

VPABSD with 128 bit operands:

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

VPABSD with 256 bit operands:

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 7th 32-bit double words
Unsigned DEST[255:224] ← ABS(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

PABSB: __m128i _mm_abs_epi8 (__m128i a)

VPABSB: __m128i _mm_abs_epi8 (__m128i a)

VPABSB: __m256i _mm256_abs_epi8 (__m256i a)

PABSW: __m128i _mm_abs_epi16 (__m128i a)

VPABSW: __m128i _mm_abs_epi16 (__m128i a)

VPABSW: __m256i _mm256_abs_epi16 (__m256i a)

PABSD: __m128i _mm_abs_epi32 (__m128i a)

VPABSD: __m128i _mm_abs_epi32 (__m128i a)

VPABSD: __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PACKSSWB/PACKSSDW — Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 63 /r PACKSSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xmm1* using signed saturation.
66 0F 6B /r PACKSSDW *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 4 packed signed doubleword integers from *xmm1* and from *xmm2/m128* into 8 packed signed word integers in xmm1 using signed saturation.
VEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Converts 8 packed signed word integers from *xmm2* and from *xmm3/m128* into 16 packed signed byte integers in *xmm1* using signed saturation.
VEX.NDS.128.66.0F.WIG 6B /r VPACKSSDW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Converts 4 packed signed doubleword integers from *xmm2* and from *xmm3/m128* into 8 packed signed word integers in xmm1 using signed saturation.
VEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 16 packed signed word integers from *ymm2* and from *ymm3/m256* into 32 packed signed byte integers in *ymm1* using signed saturation.
VEX.NDS.256.66.0F.WIG 6B /r VPACKSSDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 8 packed signed doubleword integers from *ymm2* and from *ymm3/m256* into 16 packed signed word integers in ymm1 using signed saturation.

Instruction Operand Encoding

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

Description

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

The PACKSSDW instruction packs 4 or 8 signed doublewords from the first source operand and 4 or 8 signed doublewords from the second source operand into 8 or 16 signed words in the destination operand. 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.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PACKSSWB instruction (128-bit Legacy SSE version)

```
DEST[7:0]← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ←SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
```

DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);

PACKSSDW instruction (128-bit Legacy SSE version)

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

VPACKSSWB instruction (VEX.128 encoded version)

DEST[7:0]← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[VLMAX:128]← 0;

VPACKSSDW instruction (VEX.128 encoded version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);

DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[VLMAX:128]← 0;

VPACKSSWB instruction (VEX.256 encoded version)

DEST[7:0]← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[135:128]← SaturateSignedWordToSignedByte (SRC1[143:128]);
DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);

VPACKSSDW instruction (VEX.256 encoded version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);

```
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);
DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);
DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);
DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);
```

Intel C/C++ Compiler Intrinsic Equivalent

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

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

VPACKSSWB: __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)

VPACKSSDW: __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PACKUSDW — Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW *xmm1, xmm2/m128*	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from *xmm1* and 4 packed signed doubleword integers from *xmm2/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.NDS.128.66.0F38.WIG 2B /r VPACKUSDW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Convert 4 packed signed doubleword integers from *xmm2* and 4 packed signed doubleword integers from *xmm3/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.NDS.256.66.0F38.WIG 2B /r VPACKUSDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Convert 8 packed signed doubleword integers from *ymm2* and 8 packed signed doubleword integers from *ymm3/m128* into 16 packed unsigned word integers in *ymm1* using unsigned saturation.

Instruction Operand Encoding

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

Description

Converts packed signed doubleword integers into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PACKUSDW (Legacy SSE instruction)

```
TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] ← (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] ← (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[63:48] ← (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] ← (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] ← (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] ← (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
```

PACKUSDW (VEX.128 encoded version)

```
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[63:48] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
```

```
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[VLMAX:128] ← 0;
```

VPACKUSDW (VEX.256 encoded version)

```
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[63:48] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[128:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
TMP[207:192] ← (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] ← (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PACKUSDW: __m128i _mm_packus_epi32(__m128i m1, __m128i m2);

VPACKUSDW: __m256i _mm256_packus_epi32(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PACKUSWB — Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 67 /r PACKUSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Converts 8 signed word integers from *xmm2* and 8 signed word integers from *xmm3/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 16 signed word integers from *ymm2* and 16signed word integers from *ymm3/m256* into 32 unsigned byte integers in *ymm1* using unsigned saturation.

Instruction Operand Encoding

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

Description

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

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

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

operand is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PACKUSWB (Legacy SSE instruction)

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

PACKUSWB (VEX.128 encoded version)

```
DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
```

```
    DEST[VLMAX:128] ← 0;
```

VPACKUSWB (VEX.256 encoded version)

```
    DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
    DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
    DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
    DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
    DEST[39:32] ←SaturateSignedWordToUnsignedByte (SRC1[79:64]);
    DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
    DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
    DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
    DEST[71:64] ←SaturateSignedWordToUnsignedByte (SRC2[15:0]);
    DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
    DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
    DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
    DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
    DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
    DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
    DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
    DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    DEST[143:136] ←SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    DEST[151:144] ←SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    DEST[159:152] ←SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PACKUSWB: __m128i _mm_packus_epi16(__m128i m1, __m128i m2);

VPACKUSWB: __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PADDB/PADDW/PADDD/PADDQ — Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F FC /r PADDB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
66 0F D4/r PADDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed quadword integers from *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed word integers from *xmm2, xmm3/m128* and *store in xmm1*.
VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed quadword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1.*
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1.*

Instruction Operand Encoding

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

Description

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer result in destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer result in destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand.

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer result in destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand.

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer result in destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a

carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PADDB (Legacy SSE instruction)

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

PADDW (Legacy SSE instruction)

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

PADDD (Legacy SSE instruction)

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

PADDQ (Legacy SSE instruction)

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

VPADDB (VEX.128 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 14th byte *)
DEST[127:120]← SRC1[127:120] + SRC2[127:120];
DEST[VLMAX:128] ← 0;
```

VPADDW (VEX.128 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
```

DEST[127:112]← SRC1[127:112] + SRC2[127:112];
DEST[VLMAX:128] ← 0;

VPADDD (VEX.128 encoded instruction)

DEST[31:0]← SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← SRC1[127:96] + SRC2[127:96];
DEST[VLMAX:128] ← 0;

VPADDQ (VEX.128 encoded instruction)

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

VPADDB (VEX.256 encoded instruction)

DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248]← SRC1[255:248] + SRC2[255:248];

VPADDW (VEX.256 encoded instruction)

DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)
DEST[255:240]← SRC1[255:240] + SRC2[255:240];

VPADDD (VEX.256 encoded instruction)

DEST[31:0]← SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)
DEST[255:224] ← SRC1[255:224] + SRC2[255:224];

VPADDQ (VEX.256 encoded instruction)

DEST[63:0]← SRC1[63:0] + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[191:128]← SRC1[191:128] + SRC2[191:128];
DEST[255:192] ← SRC1[255:192] + SRC2[255:192];

Intel C/C++ Compiler Intrinsic Equivalent

(V)PADDB: __m128i _mm_add_epi8 (__m128ia,__m128i b)

(V)PADDW: __m128i _mm_add_epi16 (__m128i a, __m128i b)

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

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

VPADDB: __m256i _mm256_add_epi8 (__m256ia,__m256i b)

VPADDW: __m256i _mm256_add_epi16 (__m256i a, __m256i b)

VPADDD: __m256i _mm256_add_epi32 (__m256i a, __m256i b)

VPADDQ: __m256i _mm256_add_epi64 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PADDSB/PADDSW — Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F EC /r PADDSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed byte integers from *xmm2/m128* and *xmm1* and saturate the results.
66 0F ED /r PADDSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed signed byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed signed word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1.
VEX.NDS.256.66.0F.WIG EC /r VPADDSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1.
VEX.NDS.256.66.0F.WIG ED /r VPADDSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1.

Instruction Operand Encoding

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

Description

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the

range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PADDSB (Legacy SSE instruction)

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

PADDSW (Legacy SSE instruction)

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

VPADDSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120]← SaturateToSignedByte (SRC1[127:120] + SRC2[127:120]);
DEST[VLMAX:128]← 0
```

VPADDSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112])
DEST[VLMAX:128]← 0
```

VPADDSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalent

PADDSB: __m128i _mm_adds_epi8 (__m128i a, __m128i b)

PADDSW: __m128i _mm_adds_epi16 (__m128i a, __m128i b)

VPADDSB: __m128i _mm_adds_epi8 (__m128i a, __m128i b)

VPADDSW: __m128i _mm_adds_epi16 (__m128i a, __m128i b)

VPADDSB: __m256i _mm256_adds_epi8 (__m256i a, __m256i b)

VPADDSW: __m256i _mm256_adds_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

PADDUSB/PADDUSW — Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F DC /r PADDUSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* and saturate the results.
66 0F DD /r PADDUSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.NDS.128.66.0F.WIG DC /r VPADDUSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed unsigned byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed unsigned word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed unsigned byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1.
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed unsigned word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1.

Instruction Operand Encoding

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

Description

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond

the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PADDUSB (Legacy SSE instruction)

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

PADDUSW (Legacy SSE instruction)

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

VPADDUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120]← SaturateToUnsignedByte (SRC1[127:120] + SRC2[127:120]);
DEST[VLMAX:128]← 0
```

VPADDUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112])
DEST[VLMAX:128]← 0
```

VPADDUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDUSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PADDUSB: __m128i _mm_adds_epu8 (__m128i a, __m128i b)

(V)PADDUSW: __m128i _mm_adds_epu16 (__m128i a, __m128i b)

VPADDUSB: __m256i _mm256_adds_epu8 (__m256i a, __m256i b)

VPADDUSW: __m256i _mm256_adds_epu16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PALIGNR — Byte Align

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 3A 0F /r ib PALIGNR xmm1, xmm2/m128, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
VEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
VEX.NDS.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1

Instruction Operand Encoding

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

Description

(V)PALIGNR concatenates two blocks of 16-byte data from the first source operand and the second source operand into an intermediate 32-byte composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right aligned 16-byte result into the destination. The immediate value is considered unsigned. Immediate shift counts larger than 32 for 128-bit operands produces a zero result.

Legacy SSE instructions: In 64-bit mode use the REX prefix to access additional registers.

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

VEX.256 encoded version: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand.

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.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

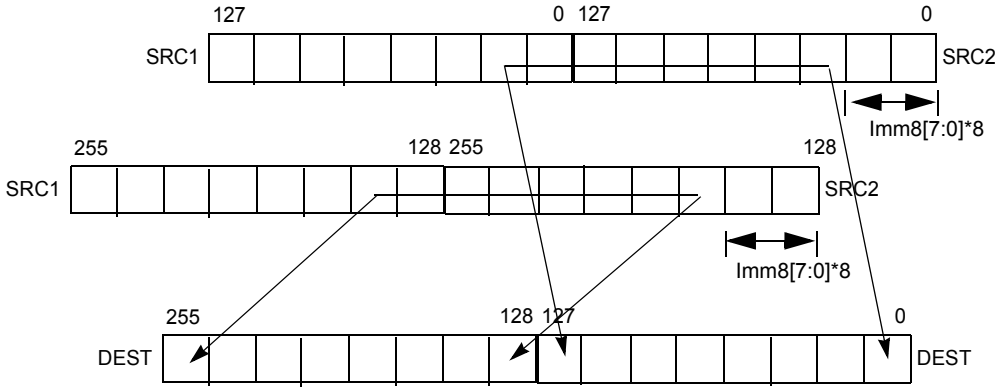


Figure 5-2. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR

temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]

DEST[VLMAX:128] (Unmodified)
VPALIGNR (VEX.128 encoded version)
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX:128] ← 0

VPALIGNR (VEX.256 encoded version)
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[255:128] ← temp1[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

(V)PALIGNR: __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)

VPALIGNR: __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PAND — Logical AND

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F DB /r PAND xmm1, xmm2/.m128	A	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/.m128	B	V/V	AVX	Bitwise AND of xmm2, and xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/.m256	B	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PAND (Legacy SSE instruction)

DEST[127:0] ← (DEST[127:0] AND SRC[127:0])

VPAND (VEX.128 encoded instruction)
DEST[127:0] ← (SRC1[127:0] AND SRC2[127:0])
DEST[VLMAX:128] ← 0

VPAND (VEX.256 encoded instruction)
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])

Intel C/C++ Compiler Intrinsic Equivalent

(V)PAND: __m128i _mm_and_si128 (__m128i a, __m128i b)

VPAND: __m256i _mm256_and_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PANDN — Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F DF /r PANDN xmm1, xmm2/.m128	A	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/.m128	B	V/V	AVX	Bitwise AND NOT of xmm2, and xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F.WIG DF /r VPANDN ymm1, ymm2, ymm3/.m256	B	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.

Instruction Operand Encoding

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

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PANDN (Legacy SSE instruction)

DEST[127:0] ← ((NOT DEST[127:0]) AND SRC[127:0])

VPANDN (VEX.128 encoded instruction)
DEST[127:0] ← ((NOT SRC1[127:0]) AND SRC2[127:0])
DEST[VLMAX:128] ← 0

VPANDN (VEX.256 encoded instruction)
DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])

Intel C/C++ Compiler Intrinsic Equivalent

(V)PANDN: __m128i _mm_andnot_si128 (__m128i a, __m128i b)

VPANDN: __m256i _mm256_andnot_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PAVGB/PAVGW — Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F E0, /r PAVGB *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
66 0F E3, /r PAVGW *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.
VEX.NDS.128.66.0F.WIG E0 /r VPAVGB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Average packed unsigned byte integers from xmm2, and *xmm3/m128* with rounding and store to xmm1.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Average packed unsigned word integers from xmm2, *xmm3/m128* with rounding to xmm1.
VEX.NDS.256.66.0F.WIG E0 /r VPAVGB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to ymm1.
VEX.NDS.256.66.0F.WIG E3 /r VPAVGW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Average packed unsigned word integers from ymm2, *ymm3/m256* with rounding to ymm1.

Instruction Operand Encoding

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

Description

Performs a SIMD average of the packed unsigned integers from the second source operand and the first operand, and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the

elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

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

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

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 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

PAVGB (Legacy SSE instruction)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 15 )
SRC[63:56] ← (SRC[127:120] + DEST[127:120)] + 1) >> 1;
```

PAVGW (Legacy SSE instruction)

```
SRC[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 7)
DEST[127:48] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 encoded instruction)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 15)
DEST[127:48] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1;
DEST[VLMAX:128] ← 0
```

VPAVGW (VEX.128 encoded instruction)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 7)
DEST[127:4] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1;
DEST[VLMAX:128] ← 0
```

VPAVGB (VEX.256 encoded instruction)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 31)
```

```
DEST[255:248] ← (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 encoded instruction)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 15)
DEST[255:14]) ← (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PAVGB: __m128i _mm_avg_epu8 (__m128i a, __m128i b)

(V)PAVGW: __m128i _mm_avg_epu16 (__m128i a, __m128i b)

VPAVGB: __m256i _mm256_avg_epu8 (__m256i a, __m256i b)

VPAVGW: __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PBLENDVB — Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB xmm1, xmm2/m128, <XMM0>	A	V/V	SSE4_1	Select byte values from xmm1 and xmm2/m128 from mask specified in the high bit of each byte in XMM0 and store the values into xmm1.
VEX.NDS.128.66.0F3A.W0 4C /r /is4 VPBLENDVB xmm1, xmm2, xmm3/m128, xmm4	B	V/V	AVX	Select byte values from xmm2 and xmm3/m128 from mask specified in the high bit of each byte in xmm4 and store the values into xmm1.
VEX.NDS.256.66.0F3A.W0 4C /r /is4 VPBLENDVB ymm1, ymm2, ymm3/m256, ymm4	B	V/V	AVX2	Select byte values from ymm2 and ymm3/m256 from mask specified in the high bit of each byte in ymm4 and store the values into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	imm[3:0](r)

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.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

VPBLENDVB (VEX.256 encoded version)

```
MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ? SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ? SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ? SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ? SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ? SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ? SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ? SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ? SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ? SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ? SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ? SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
```

```
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
IF (MASK[135] == 1) THEN DEST[135:128] ← SRC2[135:128];
ELSE DEST[135:128] ← SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144]
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152]
ELSE DEST[159:152] ← SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] ← SRC2[167:160]
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] ← SRC2[183:176]
ELSE DEST[183:176] ← SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] ← SRC2[191:184]
ELSE DEST[191:184] ← SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] ← SRC2[199:192]
ELSE DEST[199:192] ← SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] ← SRC2[207:200]
ELSE DEST[207:200] ← SRC1[207:200]
IF (MASK[215] == 1) THEN DEST[215:208] ← SRC2[215:208]
ELSE DEST[215:208] ← SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] ← SRC2[223:216]
ELSE DEST[223:216] ← SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] ← SRC2[231:224]
ELSE DEST[231:224] ← SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] ← SRC2[239:232]
ELSE DEST[239:232] ← SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] ← SRC2[247:240]
ELSE DEST[247:240] ← SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] ← SRC2[255:248]
ELSE DEST[255:248] ← SRC1[255:248]
```

VPBLENDVB (VEX.128 encoded version)

```
MASK ← XMM0
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
DEST[VLMAX:128] ← 0
```

PBLENDVB (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC[7:0];
ELSE DEST[7:0] ← DEST[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC[15:8];
ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC[23:16]
```

```
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120])

DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V) PBLENDVB: __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);

VPBLENDVB: __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally,

#UD	If VEX.W = 1.

PBLENDW — Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW xmm1, xmm2/m128, imm8	A	V/V	SSE4_1	Select words from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.
VEX.NDS.128.66.0F3A.WIG 0E /r ib VPBLENDW xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select words from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.
VEX.NDS.256.66.0F3A.WIG 0E /r ib VPBLENDW ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX2	Select words from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.

Instruction Operand Encoding

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

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

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

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

Operation

VPBLENDW (VEX.256 encoded version)

```
IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
IF (imm8[0] == 1) THEN DEST[143:128] ← SRC2[143:128]
ELSE DEST[143:128] ← SRC1[143:128]
IF (imm8[1] == 1) THEN DEST[159:144] ← SRC2[159:144]
ELSE DEST[159:144] ← SRC1[159:144]
IF (imm8[2] == 1) THEN DEST[175:160] ← SRC2[175:160]
ELSE DEST[175:160] ← SRC1[175:160]
IF (imm8[3] == 1) THEN DEST[191:176] ← SRC2[191:176]
ELSE DEST[191:176] ← SRC1[191:176]
IF (imm8[4] == 1) THEN DEST[207:192] ← SRC2[207:192]
ELSE DEST[207:192] ← SRC1[207:192]
IF (imm8[5] == 1) THEN DEST[223:208] ← SRC2[223:208]
ELSE DEST[223:208] ← SRC1[223:208]
IF (imm8[6] == 1) THEN DEST[239:224] ← SRC2[239:224]
ELSE DEST[239:224] ← SRC1[239:224]
IF (imm8[7] == 1) THEN DEST[255:240] ← SRC2[255:240]
ELSE DEST[255:240] ← SRC1[255:240]
```

VPBLENDW (VEX.128 encoded version)

```
IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
```

ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
DEST[VLMAX: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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW: __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask)

VPBLENDW: __m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ — Compare Packed Integers for Equality

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 74 /r PCMPEQB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed bytes in xmm2/m128 and xmm1 for equality.
66 0F 75 /r PCMPEQW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed words in xmm2/m128 and xmm1 for equality.
66 0F 76 /r PCMPEQD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed doublewords in xmm2/m128 and xmm1 for equality.
66 0F 38 29 /r PCMPEQQ xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed quadwords in xmm2/m128 and xmm1 for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3 /m128	B	V/V	AVX	Compare packed bytes in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed words in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed doublewords in xmm3/m128 and xmm2 for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed quadwords in xmm3/m128 and xmm2 for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed words in ymm3/m256 and ymm2 for equality.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed doublewords in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed quadwords in ymm3/m256 and ymm2 for equality.

Instruction Operand Encoding

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

Description

Performs a SIMD compare for equality of the packed bytes, words, doublewords, or quadwords in the first source operand and the second source operand. If a pair of data elements is equal the corresponding data element in the destination operand is set to all 1s, otherwise it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands, and the (V)PCMPEQQ instruction compares the corresponding quadwords in the destination and source operands.

Legacy SSE instructions: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. In 64-bit mode using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

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

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

Operation

```
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.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
```

VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0
```

PCMPEQB (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

VPCMPEQW (VEX.256 encoded version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])

VPCMPEQW (VEX.128 encoded version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPEQW (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

VPCMPEQD (VEX.256 encoded version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])

VPCMPEQD (VEX.128 encoded version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPEQD (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

VPCMPEQQ (VEX.256 encoded version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])

VPCMPEQQ (VEX.128 encoded version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPEQQ (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_QWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCMPEQB: __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)

(V)PCMPEQW: __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)

(V)PCMPEQD: __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)

(V)PCMPEQQ: __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);

VPCMPEQB: __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)

VPCMPEQW: __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)

VPCMPEQD: __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

VPCMPEQQ: __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ — Compare Packed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 64 /r PCMPGTB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.
66 0F 65 /r PCMPGTW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.
66 0F 66 /r PCMPGTD xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed double-word integers in xmm1 and xmm2/m128 for greater than.
66 0F 38 37 /r PCMPGTQ xmm1, xmm2/m128	A	V/V	SSE4_2	Compare packed qwords in xmm2/m128 and xmm1 for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed double-word integers in xmm2 and xmm3/m128 for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed qwords in xmm2 and xmm3/m128 for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed double-word integers in ymm2 and ymm3/m256 for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed qwords in ymm2 and ymm3/m256 for greater than.

Instruction Operand Encoding

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

Description

Performs a SIMD signed compare for the greater value of the packed byte, word, doubleword, or quadword integers in the first source operand and the second source operand. If a data element in the first source operand is greater than the corresponding date 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 (V)PCMPGTB instruction compares the corresponding signed byte integers in the first and second source operands; the (V)PCMPGTW instruction compares the corresponding signed word integers in the first and second source operands; the (V)PCMPGTD instruction compares the corresponding signed doubleword integers in the first and second source operands, and the (V)PCMPGTQ instruction compares the corresponding signed qword integers in the first and second source operands.

Legacy SSE instructions: In 64-bit mode using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15). The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers.

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

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

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

Operation

```
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.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
```

VPCMPGTB (VEX.128 encoded version)
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPGTB (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

VPCMPGTW (VEX.256 encoded version)
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])

VPCMPGTW (VEX.128 encoded version)
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPGTW (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

VPCMPGTD (VEX.256 encoded version)
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])

VPCMPGTD (VEX.128 encoded version)
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPGTD (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[VLMAX:128] (Unmodified)

VPCMPGTQ (VEX.256 encoded version)
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])

VPCMPGTQ (VEX.128 encoded version)
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[VLMAX:128] ← 0

PCMPGTQ (128-bit Legacy SSE version)
DEST[127:0] ←COMPARE_QWORDS_GREATER(DEST[127:0],SRC2[127:0])
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCMPGTB: __m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)

(V)PCMPGTW: __m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)

(V)PCMPGTD: __m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)

(V)PCMPGTQ: __m128i _mm_cmpgt_epi64(__m128i a, __m128i b);

VPCMPGTB: __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)

VPCMPGTW: __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)

VPCMPGTD: __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

VPCMPGTQ: __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PHADDW/PHADDD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 01 /r PHADDW xmm1, xmm2/m128	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack to xmm1.
66 0F 38 02 /r PHADDD xmm1, xmm2/m128	A	V/V	SSSE3	Add 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add 16-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 02 /r VPHADDD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.256.66.0F38.WIG 01 /r VPHADDW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add 16-bit signed integers horizontally, pack to ymm1.
VEX.NDS.256.66.0F38.WIG 02 /r VPHADDD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add 32-bit signed integers horizontally, pack to ymm1.

Instruction Operand Encoding

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

Description

(V)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. (V)PHADDD 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.

Legacy SSE instructions: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. In 64-bit mode use the REX prefix to access additional registers.

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

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

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

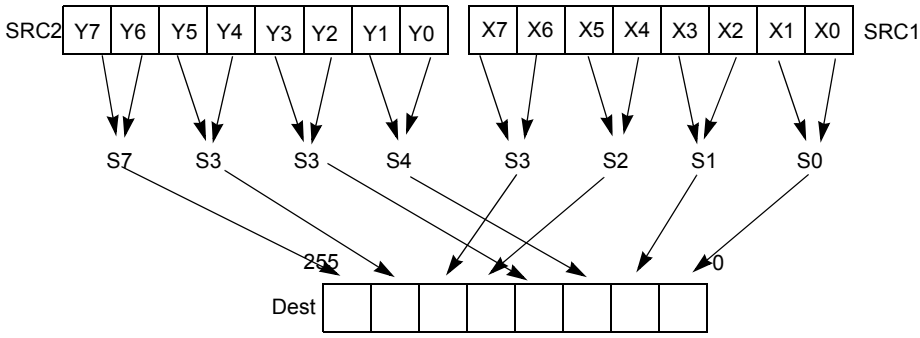


Figure 5-3. 256-bit VPHADDD Instruction Operation

Operation

VPHADDW (VEX.256 encoded version)

DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]

DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[143:128] ← SRC1[159:144] + SRC1[143:128]
DEST[159:144] ← SRC1[191:176] + SRC1[175:160]
DEST[175:160] ← SRC1[223:208] + SRC1[207:192]
DEST[191:176] ← SRC1[255:240] + SRC1[239:224]
DEST[207:192] ← SRC2[127:112] + SRC2[143:128]
DEST[223:208] ← SRC2[159:144] + SRC2[175:160]
DEST[239:224] ← SRC2[191:176] + SRC2[207:192]
DEST[255:240] ← SRC2[223:208] + SRC2[239:224]

VPHADDD (VEX.256 encoded version)
DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[159-128] ← SRC1[191-160] + SRC1[159-128]
DEST[191-160] ← SRC1[255-224] + SRC1[223-192]
DEST[223-192] ← SRC2[191-160] + SRC2[159-128]
DEST[255-224] ← SRC2[255-224] + SRC2[223-192]

VPHADDW (VEX.128 encoded version)
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[VLMAX:128] ← 0

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

PHADDD (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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PHADDW: __m128i _mm_hadd_epi16 (__m128i a, __m128i b)

(V)PHADDD: __m128i _mm_hadd_epi32 (__m128i a, __m128i b)

VPHADDW: __m256i _mm256_hadd_epi16 (__m256i a, __m256i b)

VPHADDD: __m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PHADDSW — Packed Horizontal Add with Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 03 /r PHADDSW xmm1, xmm2/m128	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to xmm1.
VEX.NDS.256.66.0F38.WIG 03 /r VPHADDSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to ymm1.

Instruction Operand Encoding

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

Description

(V)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.

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

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

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

Operation

VPHADDSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
```

DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128]= SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord(SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191-160] + SRC2[159-128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])

VPHADDSW (VEX.128 encoded version)
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[VLMAX: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])
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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PHADDSW: __m128i _mm_hadds_epi16 (__m128i a, __m128i b)

VPHADDSW: __m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 05 /r PHSUBW xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to xmm1.
66 0F 38 06 /r PHSUBD xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to xmm1.
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to xmm1.
VEX.NDS.256.66.0F38.WIG 05 /r VPHSUBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to ymm1.
VEX.NDS.256.66.0F38.WIG 06 /r VPHSUBD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to ymm1.

Instruction Operand Encoding

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

Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the second source operand and destination operands, and packs the signed 16-bit results to the destination operand. (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory loca-

tion. Bits (255:128) of the corresponding YMM destination register remain unchanged.

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

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

Operation

VPHSUBW (VEX.256 encoded version)
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]

VPHSUBD (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC1[191:160]
DEST[191:160] ← SRC1[223:192] - SRC1[255:224]
DEST[223:192] ← SRC2[159:128] - SRC2[191:160]
DEST[255:224] ← SRC2[223:192] - SRC2[255:224]

VPHSUBW (VEX.128 encoded version)
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]

DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[VLMAX: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[VLMAX: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[VLMAX: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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PHSUBW: __m128i _mm_hsub_epi16 (__m128i a, __m128i b)

(V)PHSUBD: __m128i _mm_hsub_epi32 (__m128i a, __m128i b)

VPHSUBW: __m256i _mm256_hsub_epi16 (__m256i a, __m256i b)

VPHSUBD: __m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PHSUBSW — Packed Horizontal Subtract with Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 07 /r PHSUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.
VEX.NDS.256.66.0F38.WIG 07 /r VPHSUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to ymm1.

Instruction Operand Encoding

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

Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the 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: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

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

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

Operation

VPHSUBSW (VEX.256 encoded version)
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128]= SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])

VPHSUBSW (VEX.128 encoded version)
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[VLMAX: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]
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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PHSUBSW: __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)

VPHSUBSW: __m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMADDUBSW — Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1.

Instruction Operand Encoding

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

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the first source operand with the corresponding signed byte of the second source operand, producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7:0) in the first source and second source operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15:8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15:0). The same operation is performed on the other pairs of adjacent bytes.

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

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

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

Operation

VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]*
SRC1[247:240])
```

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]*
SRC1[119:112])
DEST[VLMAX:128] ← 0
```

PMADDUBSW (128-bit Legacy SSE version)

```
DEST[15:0] ← SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]*
DEST[119:112]);
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMADDUBSW: __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)

VPMADDUBSW: __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMADDWD — Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F F5 /r PMADDWD xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1.

Instruction Operand Encoding

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

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15:0) and (31-16) in the second source and first source operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words.

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory loca-

tion. Bits (255:128) of the corresponding YMM destination register remain unchanged.

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

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

Operation

VPMADDWD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
```

VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[VLMAX:128] ← 0
```

PMADDWD (128-bit Legacy SSE version)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16])
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48])
DEST[95:64] ← (DEST[79:64] * SRC[79:64]) + (DEST[95:80] * SRC[95:80])
DEST[127:96] ← (DEST[111:96] * SRC[111:96]) + (DEST[127:112] * SRC[127:112])
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMADDWD: __m128i _mm_madd_epi16 (__m128i a, __m128i b)

VPMADDWD: __m256i _mm256_madd_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMAXSB/PMAXSW/PMAXSD — Maximum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m128 and store packed maximum values in ymm1.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m128 and ymm2 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed maximum values in ymm1.

Instruction Operand Encoding

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

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the 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: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (255:128) of the corresponding YMM destination register remain unchanged.

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

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

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[VLMAX: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[VLMAX:128] ← 0
```

VPMAXSB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
```

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[VLMAX:128] (Unmodified)
```

VPMAXSW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX:128] ← 0
```

VPMAXSW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
```

PMAXSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] >SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[VLMAX:128] (Unmodified)
```

VPMAXSD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] > SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
```

```
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX:128] ← 0
```

VPMAXSD (VEX.256 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] ← SRC1[255:224];
    ELSE
        DEST[255:224] ← SRC2[255:224]; FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

(V)PMAXSW: __m128i _mm_max_epi16 (__m128i a, __m128i b)

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

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

VPMAXSW: __m256i _mm256_max_epi16 (__m256i a, __m256i b)

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMAXUB/PMAXUW/PMAXUD — Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3E /r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F.WIG DE /r VPMAXUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.

Instruction Operand Encoding

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

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.

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

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

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

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[VLMAX: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[VLMAX:128] ← 0
```

VPMAXUB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
```

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[VLMAX:128] (Unmodified)
```

VPMAXUW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
```

```
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX:128] ← 0
```

VPMAXUW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
```

PMAXUD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] >SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[VLMAX:128] (Unmodified)
```

VPMAXUD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] > SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX:128] ← 0
```

VPMAXUD (VEX.256 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMAXUB: __m128i _mm_max_epu8 (__m128i a, __m128i b);

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

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

VPMAXUB: __m256i _mm256_max_epu8 (__m256i a, __m256i b);

VPMAXUW: __m256i _mm256_max_epu16 (__m256i a, __m256i b)

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMINSB/PMINSW/PMINSD — Minimum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.

Instruction Operand Encoding

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

Description

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

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

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

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

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[VLMAX: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[VLMAX:128] ← 0
```

VPMINSB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
```

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[VLMAX:128] (Unmodified)
```

VPMINSW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
```

```
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[VLMAX:128] ← 0
```

VPMINSW (VEX.256 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
```

PMINSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:95] < SRC[127:95] THEN
        DEST[127:95] ← DEST[127:95];
    ELSE
        DEST[127:95] ← SRC[127:95]; FI;
DEST[VLMAX:128] (Unmodified)
```

VPMINSD (VEX.128 encoded version)

```
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:95] < SRC2[127:95] THEN
        DEST[127:95] ← SRC1[127:95];
    ELSE
        DEST[127:95] ← SRC2[127:95]; FI;
DEST[VLMAX:128] ← 0
```

VPMINSD (VEX.256 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

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

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

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

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

VPMINSW: __m256i _mm256_min_epi16 (__m256i a, __m256i b)

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMINUB/PMINUW/PMINUD — Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F.WIG DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3A /r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F.WIG DA /r VPMINUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG 3A /r VPMINUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.

Instruction Operand Encoding

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

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.

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

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

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

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[VLMAX: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[VLMAX:128] ← 0
```

VPMINUB (VEX.256 encoded version)

```
VPMINUB instruction for 128-bit operands:
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
```

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[VLMAX: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[VLMAX:128] ← 0
```

VPMINUW (VEX.256 encoded version)

```
VPMINUW instruction for 128-bit operands:
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
```

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[VLMAX: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[VLMAX:128] ← 0
```

VPMINUD (VEX.256 encoded version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] ← SRC1[255:224];
    ELSE
        DEST[255:224] ← SRC2[255:224]; FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMINUB: __m128i _mm_min_epu8 (__m128i a, __m128i b)

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

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

VPMINUB: __m256i _mm256_min_epu8 (__m256i a, __m256i b)

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMOVMSKB — Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F D7 /r PMOVMSKB reg, xmm1	A	V/V	SSE2	Move a 16-bit mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.
VEX.128.66.0F.WIG D7 /r VPMOVMSKB reg, xmm1	A	V/V	AVX	Move a 16-bit mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMSKB reg, ymm1	A	V/V	AVX2	Move a 32-bit mask of ymm1 to reg. The upper bits of r64 are filled with zeros.

Instruction Operand Encoding

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

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low word or dword of the destination operand (first operand). The source operand is an XMM register; the destination operand is a general-purpose register.

The mask is 16-bits for 128-bit source operand and 32-bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode the default operand size of the destination operand is 64 bits. Bits 63:32 are filled with zero if the source operand is a 256-bit YMM register. The upper bits above bit 15 are filled with zeros if the source operand is a 128-bit XMM register. REX.W is ignored

VEX.128 encoded version: The source operand is XMM register.

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

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

Operation

VPMOVMSKB instruction with 256-bit source operand and r32:

r32[0] ← SRC[7];
r32[1] ← SRC[15];

(* Repeat operation for bytes 3rd through 31*)
r32[31] ← SRC[255];

VPMOVMSKB instruction with 256-bit source operand and r64:
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 31*)
r64[31] ← SRC[255];
r64[63:32] ← ZERO_FILL;

PMOVMSKB instruction with 128-bit source operand and r32:
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;

PMOVMSKB instruction with 128-bit source operand and r64:
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] ← SRC[127];
r64[63:16] ← ZERO_FILL;

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMOVMSKB: int _mm_movemask_epi8 (__m128i a)

VPMOVMSKB: int _mm256_movemask_epi8 (__m256i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7

PMOVSX — Packed Move with Sign Extend

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

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1

Instruction Operand Encoding

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

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 corresponding YMM register are zeroed.

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

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

Operation

```
Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)
DEST[15:0] ← SignExtend(SRC[7:0]);
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);

Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)
```

```
DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);

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

Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)
DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);

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

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

VPMOVSXBW (VEX.256 encoded version)

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
```

VPMOVSXBD (VEX.256 encoded version)

```
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
```

VPMOVSXBQ (VEX.256 encoded version)

```
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
```

VPMOVSXWD (VEX.256 encoded version)

```
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
```

VPMOVSXWQ (VEX.256 encoded version)

```
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
```

VPMOVSXDQ (VEX.256 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])

VPMOVSXBW (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_WORDDEST[127:0], SRC[127:0]()
DEST[VLMAX:128] ← 0

VPMOVSXBD (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] ← 0

VPMOVSXBQ (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] ← 0

VPMOVSXWD (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] ← 0

VPMOVSXWQ (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] ← 0

VPMOVSXDQ (VEX.128 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] ← 0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] (Unmodified)

PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] (Unmodified)

PMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] (Unmodified)

PMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] (Unmodified)

PMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] (Unmodified)

PMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMOVSXBW: __m128i _mm_cvtepi8_epi16 (__m128i a);

(V)PMOVSXBD: __m128i _mm_cvtepi8_epi32 (__m128i a);

(V)PMOVSXBQ: __m128i _mm_cvtepi8_epi64 (__m128i a);

(V)PMOVSXWD: __m128i _mm_cvtepi16_epi32 (__m128i a);

(V)PMOVSXWQ: __m128i _mm_cvtepi16_epi64 (__m128i a);

(V)PMOVSXDQ: __m128i _mm_cvtepi32_epi64 (__m128i a);

VPMOVSXBW: __m256i _mm256_cvtepi8_epi16 (__m128i a);

VPMOVSXBD: __m256i _mm256_cvtepi8_epi32 (__m128i a);

VPMOVSXBQ: __m256i _mm256_cvtepi8_epi64 (__m128i a);

VPMOVSXWD: __m256i _mm256_cvtepi16_epi32 (__m128i a);

VPMOVSXWQ: __m256i _mm256_cvtepi16_epi64 (__m128i a);

VPMOVSXDQ: __m256i _mm256_cvtepi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

PMOVZX — Packed Move with Zero Extend

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1
VEX.128.66.0F38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 16-bit integers in ymm1
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1

Instruction Operand Encoding

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

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 corresponding YMM register are zeroed.

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

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

Operation

```
Packed_Zero_Extend_BYTE_to_WORD(DEST, SRC)
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);

Packed_Zero_Extend_BYTE_to_DWORD(DEST, SRC)
```

```
DEST[31:0] ← ZeroExtend(SRC[7:0]);
DEST[63:32] ← ZeroExtend(SRC[15:8]);
DEST[95:64] ← ZeroExtend(SRC[23:16]);
DEST[127:96] ← ZeroExtend(SRC[31:24]);

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

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)
DEST[31:0] ← ZeroExtend(SRC[15:0]);
DEST[63:32] ← ZeroExtend(SRC[31:16]);
DEST[95:64] ← ZeroExtend(SRC[47:32]);
DEST[127:96] ← ZeroExtend(SRC[63:48]);

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

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

VPMOVZXBW (VEX.256 encoded version)

```
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
```

VPMOVZXBD (VEX.256 encoded version)

```
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
```

VPMOVZXBQ (VEX.256 encoded version)

```
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
```

VPMOVZXWD (VEX.256 encoded version)

```
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
```

VPMOVZXWQ (VEX.256 encoded version)

```
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
```

VPMOVZXDQ (VEX.256 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])

VPMOVZXBW (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_WORD()
DEST[VLMAX:128] ← 0

VPMOVZXBD (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[VLMAX:128] ← 0

VPMOVZXBQ (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[VLMAX:128] ← 0

VPMOVZXWD (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_DWORD()
DEST[VLMAX:128] ← 0

VPMOVZXWQ (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_QWORD()
DEST[VLMAX:128] ← 0

VPMOVZXDQ (VEX.128 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[VLMAX:128] ← 0

PMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[VLMAX:128] (Unmodified)

PMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[VLMAX:128] (Unmodified)

PMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[VLMAX:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[VLMAX:128] (Unmodified)

PMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[VLMAX:128] (Unmodified)

PMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMOVZXBW: __m128i _mm_cvtepu8_epi16 (__m128i a);

(V)PMOVZXBD: __m128i _mm_cvtepu8_epi32 (__m128i a);

(V)PMOVZXBQ: __m128i _mm_cvtepu8_epi64 (__m128i a);

(V)PMOVZXWD: __m128i _mm_cvtepu16_epi32 (__m128i a);

(V)PMOVZXWQ: __m128i _mm_cvtepu16_epi64 (__m128i a);

(V)PMOVZXDQ: __m128i _mm_cvtepu32_epi64 (__m128i a);

VPMOVZXBW: __m256i _mm256_cvtepu8_epi16 (__m128i a);

VPMOVZXBD: __m256i _mm256_cvtepu8_epi32 (__m128i a);

VPMOVZXBQ: __m256i _mm256_cvtepu8_epi64 (__m128i a);

VPMOVZXWD: __m256i _mm256_cvtepu16_epi32 (__m128i a);

VPMOVZXWQ: __m256i _mm256_cvtepu16_epi64 (__m128i a);

VPMOVZXDQ: __m256i _mm256_cvtepu32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5

PMULDQ — Multiply Packed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.

Instruction Operand Encoding

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

Description

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

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

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

word integers stored in an YMM register. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation.

When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

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

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

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

Operation

VPMULDQ (VEX.256 encoded version)
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[191:128] ← SRC1[159:128] * SRC2[159:128]
DEST[255:192] ← SRC1[223:192] * SRC2[223:192]

VPMULDQ (VEX.128 encoded version)
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[VLMAX:128] ← 0

PMULDQ (128-bit Legacy SSE version)
DEST[63:0] ← DEST[31:0] * SRC[31:0]
DEST[127:64] ← DEST[95:64] * SRC[95:64]
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMULHRSW — Multiply Packed Unsigned Integers with Round and Scale

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed double-words, pack high 16 bits to xmm1.
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed double-words, pack high 16 bits to xmm1.
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed double-words, pack high 16 bits to ymm1.

Instruction Operand Encoding

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

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.

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

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

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

Operation

VPMULHRSW (VEX.256 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[143:128] ← temp8[16:1]
DEST[159:144] ← temp9[16:1]
DEST[175:160] ← temp10[16:1]
DEST[191:176] ← temp11[16:1]
DEST[207:192] ← temp12[16:1]
DEST[223:208] ← temp13[16:1]
DEST[239:224] ← temp14[16:1]
DEST[255:240] ← temp15[16:1]
```

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
```

```
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX:128] ← 0
```

PMULHRSW (128-bit Legacy SSE version)

```
temp0[31:0] ← INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULHRSW: __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)

VPMULHRSW: __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMULHUW — Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F E4 /r PMULHUW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.

Instruction Operand Encoding

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

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the first source operand and the second source operand, and stores the high 16 bits of each 32-bit intermediate results in the destination operand.

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

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

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

Operation

PMULHUW (VEX.256 encoded version)

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]

DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]

PMULHUW (VEX.128 encoded version)
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]

DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX: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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULHUW: __m128i _mm_mulhi_epu16 (__m128i a, __m128i b)

VPMULHUW: __m256i _mm256_mulhi_epu16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMULHW — Multiply Packed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E5 /r PMULHW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.256.66.0F.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.

Instruction Operand Encoding

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

Description

Performs a SIMD signed multiply of the packed signed word integers in the first source operand and the second source operand, and stores the high 16 bits of each intermediate 32-bit result in the destination operand.

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

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

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

Operation

PMULHW (VEX.256 encoded version)

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]

DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]

PMULHW (VEX.128 encoded version)
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]

DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX: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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULHW: __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

VPMULHW: __m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMULLW/PMULLD — Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F D5 /r PMULLW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	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.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1
VEX.NDS.256.66.0F.WIG D5 /r VPMULLW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1
VEX.NDS.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1

Instruction Operand Encoding

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

Description

Performs a SIMD signed multiply of the packed signed word (dword) integers in the first source operand and the second source operand and stores the low 16(32) bits of each intermediate 32-bit(64-bit) result in the destination operand.

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

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

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

Operation

VPMULLD (VEX.256 encoded version)

```
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
Temp4[63:0] ← SRC1[159:128] * SRC2[159:128]
Temp5[63:0] ← SRC1[191:160] * SRC2[191:160]
Temp6[63:0] ← SRC1[223:192] * SRC2[223:192]
Temp7[63:0] ← SRC1[255:224] * SRC2[255:224]

DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[159:128] ← Temp4[31:0]
DEST[191:160] ← Temp5[31:0]
DEST[223:192] ← Temp6[31:0]
DEST[255:224] ← Temp7[31:0]
```

VPMULLD (VEX.128 encoded version)

Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[VLMAX:128] ← 0

PMULLD (128-bit Legacy SSE version)
Temp0[63:0] ← DEST[31:0] * SRC[31:0]
Temp1[63:0] ← DEST[63:32] * SRC[63:32]
Temp2[63:0] ← DEST[95:64] * SRC[95:64]
Temp3[63:0] ← DEST[127:96] * SRC[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[VLMAX:128] (Unmodified)

VPMULLW (VEX.256 encoded version)
Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
Temp8[31:0] ← SRC1[143:128] * SRC2[143:128]
Temp9[31:0] ← SRC1[159:144] * SRC2[159:144]
Temp10[31:0] ← SRC1[175:160] * SRC2[175:160]
Temp11[31:0] ← SRC1[191:176] * SRC2[191:176]
Temp12[31:0] ← SRC1[207:192] * SRC2[207:192]
Temp13[31:0] ← SRC1[223:208] * SRC2[223:208]
Temp14[31:0] ← SRC1[239:224] * SRC2[239:224]
Temp15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]

DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[143:128] ← Temp8[15:0]
DEST[159:144] ← Temp9[15:0]
DEST[175:160] ← Temp10[15:0]
DEST[191:176] ← Temp11[15:0]
DEST[207:192] ← TEMP12[15:0]
DEST[223:208] ← Temp13[15:0]
DEST[239:224] ← Temp14[15:0]
DEST[255:240] ← Temp15[15:0]

VPMULLW (VEX.128 encoded version)
Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[VLMAX:128] ← 0

PMULLW (128-bit Legacy SSE version)
Temp0[31:0] ← DEST[15:0] * SRC[15:0]
Temp1[31:0] ← DEST[31:16] * SRC[31:16]
Temp2[31:0] ← DEST[47:32] * SRC[47:32]
Temp3[31:0] ← DEST[63:48] * SRC[63:48]
Temp4[31:0] ← DEST[79:64] * SRC[79:64]
Temp5[31:0] ← DEST[95:80] * SRC[95:80]
Temp6[31:0] ← DEST[111:96] * SRC[111:96]
Temp7[31:0] ← DEST[127:112] * SRC[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]

DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[127:96] ← Temp3[31:0];
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULLW: __m128i _mm_mullo_epi16 (__m128i a, __m128i b);

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

VPMULLW: __m256i _mm256_mullo_epi16 (__m256i a, __m256i b);

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PMULUDQ — Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.

Instruction Operand Encoding

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

Description

Multiplies packed unsigned doubleword integers in the first source operand by the packed unsigned doubleword integers in second source operand and stores packed unsigned quadword results in the destination operand.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (255:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation.The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

Operation

VPMULUDQ (VEX.256 encoded version)
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64
DEST[191:128] ← SRC1[159:128] * SRC2[159:128]
DEST[255:192] ← SRC1[223:192] * SRC2[223:192]

VPMULUDQ (VEX.128 encoded version)
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[VLMAX:128] ← 0

PMULUDQ (128-bit Legacy SSE version)
DEST[63:0] ← DEST[31:0] * SRC[31:0]
DEST[127:64] ← DEST[95:64] * SRC[95:64]
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PMULUDQ: __m128i _mm_mul_epu32(__m128i a, __m128i b);

VPMULUDQ: __m256i _mm256_mul_epu32(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

POR — Bitwise Logical Or

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F EB /r POR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.
VEX.NDS.256.66.0F.WIG EB /r VPOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise OR of ymm2/m256 and ymm3.

Instruction Operand Encoding

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

Description

Performs a bitwise logical OR operation on the second source operand and the first source operand and stores the result in the destination operand. Each bit of the result is set to 1 if either of the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

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

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

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

Operation

VPOR (VEX.256 encoded version)
DEST ← SRC1 OR SRC2

VPOR (VEX.128 encoded version)
DEST[127:0] ← (SRC[127:0] OR SRC2[127:0])
DEST[VLMAX:128] ← 0

POR (128-bit Legacy SSE version)
DEST[127:0] ← (SRC[127:0] OR SRC2[127:0])
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)POR: __m128i _mm_or_si128 (__m128i a, __m128i b)

VPOR: __m256i _mm256_or_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

none

Other Exceptions

See Exceptions Type 4

PSADBW — Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F F6 /r PSADBW xmm1, xmm2/m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2 /m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

Instruction Operand Encoding

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

Description

Computes the absolute value of the difference of packed groups of 8 unsigned byte integers from the second source operand and from the first source operand. The first 8 differences are summed to produce an unsigned word integer that is stored in the low word of the destination; the second 8 differences are summed to produce an unsigned word in bits 79:64 of the destination. In case of VEX.256 encoded version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are

summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

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

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

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

Operation

VPSADBW (VEX.256 encoded version)

```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[143:128] ←SUM(TEMP16:TEMP23)
DEST[191:144] ← 000000000000H
DEST[207:192] ← SUM(TEMP24:TEMP31)
DEST[223:208] ← 00000000000H
```

VPSADBW (VEX.128 encoded version)

```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[VLMAX:128] ← 0
```

PSADBW (128-bit Legacy SSE version)

```
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
```

DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000

DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSADBW: __m128i _mm_sad_epu8(__m128i a, __m128i b)

VPSADBW: __m256i _mm256_sad_epu8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 00 /r PSHUFB xmm1, xmm2/m128	A	V/V	SSSE3	Shuffle bytes in xmm1 according to contents of xmm2/m128.
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Shuffle bytes in xmm2 according to contents of xmm3/m128.
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Shuffle bytes in ymm2 according to contents of ymm3/m256.

Instruction Operand Encoding

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

Description

PSHUFB performs in-place shuffles of bytes in the first source operand according to the shuffle control mask in the second source operand. The instruction permutes the data in the first source operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the first source operand. The value of each index is the least significant 4 bits of the shuffle control byte. The first source and destination operands are XMM registers. The second source is either an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: The first 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.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask. The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

Operation

VPSHUFB (VEX.256 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7..(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (VEX.128 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[VLMAX: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[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFB: __m128i _mm_shuffle_epi8(__m128i a, __m128i b)

VPSHUFB: __m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSHUFD — Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.66.0F.WIG 70 /r ib VPSHUFD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Copies doublewords from the source operand and inserts them in the destination operand at the locations selected with the immediate control operand. Figure 5-4 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 5-4) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

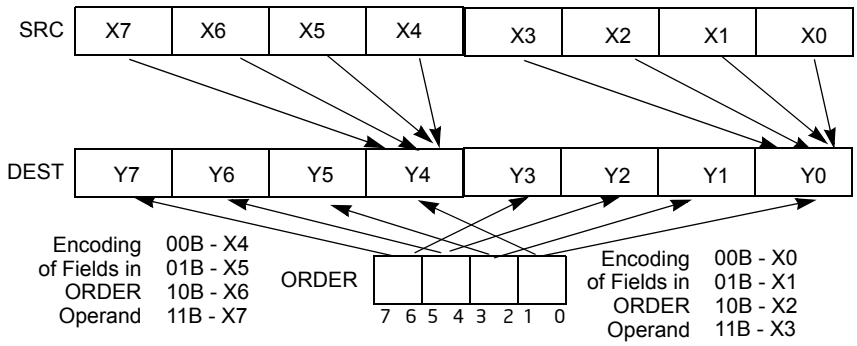


Figure 5-4. 256-bit VPSHUFD Instruction Operation

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

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

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

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

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

Operation

VPSHUFD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
```

VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX:128] ← 0
```

PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFD: __m128i _mm_shuffle_epi32(__m128i a, const int n)

VPSHUFD: __m256i _mm256_shuffle_epi32(__m256i a, const int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSHUFHW — Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand . This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 5-4. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

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

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

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

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

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

Operation

VPSHUFHW (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] ← SRC1[191:128]
DEST[207192] ← (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
```

VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[VLMAX: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[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFHW: __m128i _mm_shufflehi_epi16(__m128i a, const int n)

VPSHUFHW: __m256i _mm256_shufflehi_epi16(__m256i a, const int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSHUFLW — Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1, xmm2/m128, imm8	A	V/V	AVX	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1, ymm2/m256, imm8	A	V/V	AVX2	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 5-4. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

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

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

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

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

Operation

VPSHUFLW (VEX.256 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC1[127:64]
DEST[143:128] ← (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] ← SRC1[255:192]
```

VPSHUFLW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX:128] ← 0
```

PSHUFLW (128-bit Legacy SSE version)

```
DEST[15:0] ← (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSHUFLW: __m128i _mm_shufflelo_epi16(__m128i a, const int n)

VPSHUFLW: __m256i _mm256_shufflelo_epi16(__m256i a, const int n)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 08 /r PSIGNB xmm1, xmm2/m128	A	V/V	SSSE3	Negate packed byte integers in xmm1 if the corresponding sign in xmm2/m128 is less than zero.
66 0F 38 09 /r PSIGNW xmm1, xmm2/m128	A	V/V	SSSE3	Negate packed 16-bit integers in xmm1 if the corresponding sign in xmm2/m128 is less than zero.
66 0F 38 0A /r PSIGND xmm1, xmm2/m128	A	V/V	SSSE3	Negate packed doubleword integers in xmm1 if the corresponding sign in xmm2/m128 is less than zero.
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Negate packed byte integers in xmm2 if the corresponding sign in xmm3/m128 is less than zero.
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Negate packed 16-bit integers in xmm2 if the corresponding sign in xmm3/m128 is less than zero.
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Negate packed doubleword integers in xmm2 if the corresponding sign in xmm3/m128 is less than zero.
VEX.NDS.256.66.0F38.WIG 08 /r VPSIGNB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Negate packed byte integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG 09 /r VPSIGNW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Negate packed 16-bit integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.
VEX.NDS.256.66.0F38.WIG 0A /r VPSIGND ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Negate packed doubleword integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.

Instruction Operand Encoding

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

Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the first source operand if the sign of the corresponding data element in the second source operand is less than zero. If the sign 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.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers.

Legacy SSE instructions: In 64-bit mode use the REX prefix to access additional registers.

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

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

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

Operation

```
BYTE_SIGN_256b(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 31th bytes
    if (SRC2[255...248] < 0 )
        DEST[255...248] ← Neg(SRC1[255...248])
    else if(SRC2[255...248] == 0 )
        DEST[255...248] ← 0
    else if(SRC2[255...248] > 0 )
        DEST[255...248] ← SRC1[255...248]

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_256b(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 15th words
    if (SRC2[255..240] < 0 )
        DEST[255..240] ← Neg(SRC1[255..240])
    else if(SRC2[255..240] == 0 )
        DEST[255..240] ← 0
    else if(SRC2[255..240] > 0 )
        DEST[255..240] ← SRC1[255..240]
```

```
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_256b(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]
Repeat operation for 2nd through 7th double words
    if (SRC2[255..224] < 0 )
        DEST[255..224] ← Neg(SRC1[255..224])
    else if(SRC2[255..224] == 0 )
        DEST[255..224] ←0
    else if(SRC2[255..224] > 0 )
        DEST[255..224] ← SRC1[255..224]

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]
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.256 encoded version)
DEST[255:0] ←BYTE_SIGN_256b(SRC1, SRC2)

VPSIGNB (VEX.128 encoded version)
DEST[127:0] ←BYTE_SIGN(SRC1, SRC2)
DEST[VLMAX:128] ← 0

PSIGNB (128-bit Legacy SSE version)
DEST[127:0] ←BYTE_SIGN(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

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

VPSIGNW (VEX.128 encoded version)
DEST[127:0] ←WORD_SIGN(SRC1, SRC2)
DEST[VLMAX:128] ← 0

PSIGNW (128-bit Legacy SSE version)
DEST[127:0] ←WORD_SIGN(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

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

VPSIGND (VEX.128 encoded version)
DEST[127:0] ←DWORD_SIGN(SRC1, SRC2)
DEST[VLMAX:128] ← 0

PSIGND (128-bit Legacy SSE version)
DEST[127:0] ←DWORD_SIGN(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSIGNB: __m128i _mm_sign_epi8 (__m128i a, __m128i b)

(V)PSIGNW: __m128i _mm_sign_epi16 (__m128i a, __m128i b)

(V)PSIGND: __m128i _mm_sign_epi32 (__m128i a, __m128i b)

VPSIGNB: __m256i _mm256_sign_epi8 (__m256i a, __m256i b)

VPSIGNW: __m256i _mm256_sign_epi16 (__m256i a, __m256i b)

VPSIGND: __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSLLDQ — Byte Shift Left

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 left by imm8 bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (r, w)	NA	NA	NA
B	VEX.vvvv (w)	ModRM:r/m (R)	NA	NA

Description

Shifts the byte elements within a 128-bit lane of 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 corresponding YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

Note: In VEX encoded versions VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register.

Operation

VPSLLDQ (VEX.256 encoded version)

TEMP ← COUNT

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

VPSLLDQ (VEX.128 encoded version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[VLMAX:128] ← 0

PSLLDQ(128-bit Legacy SSE version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSLLDQ: __m128i _mm_slli_si128 (__m128i a, const int imm)

VPSLLDQ: __m256i _mm256_slli_si256 (__m256i a, const int imm)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSLLW/PSLLD/PSLLQ — Bit Shift Left

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F F1/r PSLLW xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift words in xmm1 left by imm8 while shifting in 0s.
66 0F F2 /r PSLLD xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift doublewords in xmm1 left by imm8 while shifting in 0s.
66 0F F3 /r PSLLQ xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift quadwords in xmm1 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	C	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD xmm1, xmm2, imm8	C	V/V	AVX	Shift doublewords in xmm2 left by imm8 while shifting in 0s.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ xmm1, xmm2, imm8	C	V/V	AVX	Shift quadwords in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW ymm1, ymm2, imm8	C	V/V	AVX2	Shift words in ymm2 left by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG F2 /r VPSLLD ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD ymm1, ymm2, imm8	C	V/V	AVX2	Shift doublewords in ymm2 left by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ ymm1, ymm2, imm8	C	V/V	AVX2	Shift quadwords in ymm2 left by imm8 while shifting in 0s.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s.

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The PSLLW instruction shifts each of the words in the first source operand to the left by the number of bits specified in the count operand, the PSLLD instruction shifts each of the doublewords in the first source operand, and the PSLLQ instruction shifts the quadword (or quadwords) in the first source operand.

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

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (255:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (255:128) of the corresponding YMM destination register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

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

Note: In VEX encoded versions of 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.

Operation

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

```
LOGICAL_LEFT_SHIFT_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;

LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ← ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] << COUNT);
```

```
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] ← ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (ymm, ymm, xmm/m128)

```
DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
```

VPSLLW (ymm, imm8)

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

VPSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX:128] ← 0
```

VPSLLW (xmm, imm8)

```
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX:128] ← 0
```

PSLLW (xmm, xmm, xmm/m128)

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

PSLLW (xmm, imm8)

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

VPSLLD (ymm, ymm, xmm/m128)

```
DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
```

VPSLLD (ymm, imm8)

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

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

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

PSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

PSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)

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

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

VPSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[VLMAX:128] ← 0

VPSLLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[VLMAX:128] ← 0

PSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

PSLLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSLLW: __m128i _mm_slli_epi16 (__m128i m, int count)

(V)PSLLW: __m128i _mm_sll_epi16 (__m128i m, __m128i count)

(V)PSLLD: __m128i _mm_slli_epi32 (__m128i m, int count)

(V)PSLLD: __m128i _mm_sll_epi32 (__m128i m, __m128i count)

(V)PSLLQ: __m128i _mm_slli_epi64 (__m128i m, int count)

(V)PSLLQ: __m128i _mm_sll_epi64 (__m128i m, __m128i count)

VPSLLW: __m256i _mm256_slli_epi16 (__m256i m, int count)

VPSLLW: __m256i _mm256_sll_epi16 (__m256i m, __m128i count)

VPSLLD: __m256i _mm256_slli_epi32 (__m256i m, int count)

VPSLLD: __m256i _mm256_sll_epi32 (__m256i m, __m128i count)

VPSLLQ: __m256i _mm256_slli_epi64 (__m256i m, int count)

VPSLLQ: __m256i _mm256_sll_epi64 (__m256i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

Same as Exceptions Type 4

PSRAW/PSRAD — Bit Shift Arithmetic Right

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F E1/r PSRAW xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in sign bits.
66 0F E2 /r PSRAD xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1, xmm2, imm8	C	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD xmm1, xmm2, imm8	C	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW ymm1, ymm2, imm8	C	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD ymm1, ymm2, imm8	C	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in sign bits.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. As the bits in the data elements are shifted right, the empty high-order bits are 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), or 31 (for doublewords), then the destination operand is filled with the initial value of the sign bit.

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the first source operand to the right by the number of bits specified in the count operand; the (V)PSRAD instruction shifts each of the doublewords in the first source operand.

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

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (255:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (255:128) of the corresponding YMM destination register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate.

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

Operation

```
ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    COUNT ← 15;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    COUNT ← 31;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    COUNT ← 15;
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)
    COUNT ← 31;
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 (ymm, ymm, xmm/m128)
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)

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

VPSRAW (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX:128] ← 0

VPSRAW (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX:128] ← 0

PSRAW (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

PSRAW (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)

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

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

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

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

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

PSRAD (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSRAW: __m128i _mm_srai_epi16 (__m128i m, int count)

VPSRAW: __m256i _mm256_srai_epi16 (__m256i m, int count)

(V)PSRAW: __m128i _mm_sra_epi16 (__m128i m, __m128i count)

VPSRAW: __m256i _mm256_sra_epi16 (__m256i m, __m128i count)

(V)PSRAD: __m128i _mm_srai_epi32 (__m128i m, int count)

VPSRAD: __m256i _mm256_srai_epi32 (__m256i m, int count)

(V)PSRAD: __m128i _mm_sra_epi32 (__m128i m, __m128i count)

VPSRAD: __m256i _mm256_sra_epi32 (__m256i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

Same as Exceptions Type 4

PSRLDQ — Byte Shift Right

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ xmm1, imm8	A	V/V	SSE2	Shift xmm1 right by imm8 bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8	B	V/V	AVX	Shift xmm1 right by imm8 bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2, imm8	B	V/V	AVX2	Shift ymm1 right by imm8 bytes while shifting in 0s.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (r, w)	NA	NA	NA
B	VEX.vvvv (w)	ModRM:r/m (R)	NA	NA

Description

Shifts the byte elements within a 128-bit lane of 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 corresponding YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

Note: In VEX encoded versions VEX.vvvv encodes the destination register, and VEX.B + ModRM.r/m encodes the source register.

Operation

VPSRLDQ (VEX.256 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
```

DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)

VPSRLDQ (VEX.128 encoded version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ←16; FI
DEST ← SRC >> (TEMP * 8)
DEST[VLMAX:128] ← 0

PSRLDQ(128-bit Legacy SSE version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSRLW/PSRLD/PSRLQ — Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F D1 /r PSRLW xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift words in xmm1 right by imm8 while shifting in 0s.
66 0F D2 /r PSRLD xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift doublewords in xmm1 right by imm8 while shifting in 0s.
66 0F D3 /r PSRLQ xmm1, xmm2/m128	B	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	A	V/V	SSE2	Shift quadwords in xmm1 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW xmm1, xmm2, imm8	C	V/V	AVX	Shift words in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD xmm1, xmm2, imm8	C	V/V	AVX	Shift doublewords in xmm2 right by imm8 while shifting in 0s.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ xmm1, xmm2, xmm3/m128	D	V/V	AVX	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ xmm1, xmm2, imm8	C	V/V	AVX	Shift quadwords in xmm2 right by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW ymm1, ymm2, imm8	C	V/V	AVX2	Shift words in ymm2 right by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG D2 /r VPSRLD ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD ymm1, ymm2, imm8	C	V/V	AVX2	Shift doublewords in ymm2 right by imm8 while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ ymm1, ymm2, xmm3/m128	D	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ ymm1, ymm2, imm8	C	V/V	AVX2	Shift quadwords in ymm2 right by imm8 while shifting in 0s.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s.

The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the second source operand is a memory address, 128 bits are loaded. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

The PSRLW instruction shifts each of the words in the first source operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the doublewords in the first source operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the first source operand.

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

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

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

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

Note: In VEX encoded versions of 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.

Operation

LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ← ZeroExtend(SRC[255:240] >> COUNT);
FI;
```

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)

```
COUNT ← COUNT_SRC[63:0];
```

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

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] >> COUNT);
FI;
```

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] >> COUNT);
FI;
```

LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ← ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] >> COUNT);
FI;
```

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
FI;

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

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

VPSRLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX:128] ← 0

VPSRLW (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX:128] ← 0

PSRLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

PSRLW (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)

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

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

VPSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX:128] ← 0

VPSRLD (xmm, imm8)

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

PSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

PSRLD (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)

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

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

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

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

PSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX:128] (Unmodified)

PSRLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSRLW: __m128i _mm_srli_epi16 (__m128i m, int count)

VPSRLW: __m256i _mm256_srli_epi16 (__m256i m, int count)

(V)PSRLW: __m128i _mm_srl_epi16 (__m128i m, __m128i count)

VPSRLW: __m256i _mm256_srl_epi16 (__m256i m, __m128i count)

(V)PSRLD: __m128i _mm_srli_epi32 (__m128i m, int count)

VPSRLD: __m256i _mm256_srli_epi32 (__m256i m, int count)

(V)PSRLD: __m128i _mm_srl_epi32 (__m128i m, __m128i count)

VPSRLD: __m256i _mm256_srl_epi32 (__m256i m, __m128i count)

(V)PSRLQ: __m128i _mm_srli_epi64 (__m128i m, int count)

VPSRLQ: __m256i _mm256_srli_epi64 (__m256i m, int count)

(V)PSRLQ: __m128i _mm_srl_epi64 (__m128i m, __m128i count)

VPSRLQ: __m256i _mm256_srl_epi64 (__m256i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

Same as Exceptions Type 4

PSUBB/PSUBW/PSUBD/PSUBQ — Packed Integer Subtract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F F8 /r PSUBB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed byte integers in xmm2/m128 from xmm1.
66 0F F9 /r PSUBW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed word integers in xmm2/m128 from xmm1.
66 0F FA /r PSUBD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed doubleword integers in xmm2/m128 from xmm1.
66 0F FB/r PSUBQ xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed quadword integers in xmm2/m128 from xmm1.
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed byte integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed word integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG FA /r VPSUBD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed doubleword integers in xmm3/m128 from xmm2.
VEX.NDS.128.66.0F.WIG FB /r VPSUBQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed byte integers in ymm3/m256 from ymm2.
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed word integers in ymm3/m256 from ymm2.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG FA /r VPSUBD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed doubleword integers in ymm3/m256 from ymm2.
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.

Instruction Operand Encoding

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

Description

Subtracts the packed byte, word, doubleword, or quadword integers in the second source operand from the first source operand and stores the result in the destination operand. When a result is too large to be represented in the 8/16/32/64 integer (overflow), the result is wrapped around and the low bits are written to the destination element (that is, the carry is ignored).

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

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

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (127:128) of the corresponding YMM register are zeroed.

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

Operation

VPSUBB (VEX.256 encoded version)
DEST[7:0] ← SRC1[7:0]-SRC2[7:0]

DEST[15:8] ← SRC1[15:8]-SRC2[15:8]
DEST[23:16] ← SRC1[23:16]-SRC2[23:16]
DEST[31:24] ← SRC1[31:24]-SRC2[31:24]
DEST[39:32] ← SRC1[39:32]-SRC2[39:32]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[55:48] ← SRC1[55:48]-SRC2[55:48]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[119:112] ← SRC1[119:112]-SRC2[119:112]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[135:128] ← SRC1[135:128]-SRC2[135:128]
DEST[143:136] ← SRC1[143:136]-SRC2[143:136]
DEST[151:144] ← SRC1[151:144]-SRC2[151:144]
DEST[159:152] ← SRC1[159:152]-SRC2[159:152]
DEST[167:160] ← SRC1[167:160]-SRC2[167:160]
DEST[175:168] ← SRC1[175:168]-SRC2[175:168]
DEST[183:176] ← SRC1[183:176]-SRC2[183:176]
DEST[191:184] ← SRC1[191:184]-SRC2[191:184]
DEST[199:192] ← SRC1[199:192]-SRC2[199:192]
DEST[207:200] ← SRC1[207:200]-SRC2[207:200]
DEST[215:208] ← SRC1[215:208]-SRC2[215:208]
DEST[223:216] ← SRC1[223:216]-SRC2[223:216]
DEST[231:224] ← SRC1[231:224]-SRC2[231:224]
DEST[239:232] ← SRC1[239:232]-SRC2[239:232]
DEST[247:240] ← SRC1[247:240]-SRC2[247:240]
DEST[255:248] ← SRC1[255:248]-SRC2[255:248]

VPSUBB (VEX.128 encoded version)
DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[15:8] ← SRC1[15:8]-SRC2[15:8]
DEST[23:16] ← SRC1[23:16]-SRC2[23:16]
DEST[31:24] ← SRC1[31:24]-SRC2[31:24]
DEST[39:32] ← SRC1[39:32]-SRC2[39:32]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[55:48] ← SRC1[55:48]-SRC2[55:48]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]

DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[119:112] ← SRC1[119:112]-SRC2[119:112]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[VLMAX:128] ← 0

PSUBB (128-bit Legacy SSE version)
DEST[7:0] ← DEST[7:0]-SRC[7:0]
DEST[15:8] ← DEST[15:8]-SRC[15:8]
DEST[23:16] ← DEST[23:16]-SRC[23:16]
DEST[31:24] ← DEST[31:24]-SRC[31:24]
DEST[39:32] ← DEST[39:32]-SRC[39:32]
DEST[47:40] ← DEST[47:40]-SRC[47:40]
DEST[55:48] ← DEST[55:48]-SRC[55:48]
DEST[63:56] ← DEST[63:56]-SRC[63:56]
DEST[71:64] ← DEST[71:64]-SRC[71:64]
DEST[79:72] ← DEST[79:72]-SRC[79:72]
DEST[87:80] ← DEST[87:80]-SRC[87:80]
DEST[95:88] ← DEST[95:88]-SRC[95:88]
DEST[103:96] ← DEST[103:96]-SRC[103:96]
DEST[111:104] ← DEST[111:104]-SRC[111:104]
DEST[119:112] ← DEST[119:112]-SRC[119:112]
DEST[127:120] ← DEST[127:120]-SRC[127:120]
DEST[VLMAX:128] (Unmodified)

VPSUBW (VEX.256 encoded version)
DEST[15:0] ← SRC1[15:0]-SRC2[15:0]
DEST[31:16] ← SRC1[31:16]-SRC2[31:16]
DEST[47:32] ← SRC1[47:32]-SRC2[47:32]
DEST[63:48] ← SRC1[63:48]-SRC2[63:48]
DEST[79:64] ← SRC1[79:64]-SRC2[79:64]
DEST[95:80] ← SRC1[95:80]-SRC2[95:80]
DEST[111:96] ← SRC1[111:96]-SRC2[111:96]
DEST[127:112] ← SRC1[127:112]-SRC2[127:112]
DEST[143:128] ← SRC1[143:128]-SRC2[143:128]
DEST[159:144] ← SRC1[159:144]-SRC2[159:144]
DEST[175:160] ← SRC1[175:160]-SRC2[175:160]
DEST[191:176] ← SRC1[191:176]-SRC2[191:176]
DEST[207:192] ← SRC1207:192]-SRC2[207:192]
DEST[223:208] ← SRC1[223:208]-SRC2[223:208]
DEST[239:224] ← SRC1[239:224]-SRC2[239:224]

DEST[255:240] ← SRC1[255:240]-SRC2[255:240]

VPSUBW (VEX.128 encoded version)
DEST[15:0] ← SRC1[15:0]-SRC2[15:0]
DEST[31:16] ← SRC1[31:16]-SRC2[31:16]
DEST[47:32] ← SRC1[47:32]-SRC2[47:32]
DEST[63:48] ← SRC1[63:48]-SRC2[63:48]
DEST[79:64] ← SRC1[79:64]-SRC2[79:64]
DEST[95:80] ← SRC1[95:80]-SRC2[95:80]
DEST[111:96] ← SRC1[111:96]-SRC2[111:96]
DEST[127:112] ← SRC1[127:112]-SRC2[127:112]
DEST[VLMAX: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[VLMAX: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[159:128] ← SRC1[159:128]-SRC2[159:128]
DEST[191:160] ← SRC1[191:160]-SRC2[191:160]
DEST[223:192] ← SRC1[223:192]-SRC2[223:192]
DEST[255:224] ← SRC1[255:224]-SRC2[255:224]

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

VPSUBQ (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[191:128] ← SRC1[191:128]-SRC2[191:128]
DEST[255:192] ← SRC1[255:192]-SRC2[255:192]

VPSUBQ (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[VLMAX: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[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSUBB: __m128i _mm_sub_epi8 (__m128i a, __m128i b)

(V)PSUBW: __m128i _mm_sub_epi16 (__m128i a, __m128i b)

(V)PSUBD: __m128i _mm_sub_epi32 (__m128i a, __m128i b)

(V)PSUBQ: __m128i _mm_sub_epi64(__m128i m1, __m128i m2)

VPSUBB: __m256i _mm256_sub_epi8 (__m256i a, __m256i b)

VPSUBW: __m256i _mm256_sub_epi16 (__m256i a, __m256i b)

VPSUBD: __m256i _mm256_sub_epi32 (__m256i a, __m256i b)

VPSUBQ: __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSUBSB/PSUBSW — Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F E8 /r PSUBSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.
66 0F E9 /r PSUBSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.

Instruction Operand Encoding

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

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

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

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

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (127:128) of the corresponding YMM register are zeroed.

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

Operation

VPSUBSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
```

VPSUBSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[VLMAX:128] ← 0
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[VLMAX:128] (Unmodified)
```

VPSUBSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);

VPSUBSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[VLMAX:128] ← 0

PSUBSW (128-bit Legacy SSE Version)
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSUBSB: __m128i _mm_subs_epi8(__m128i m1, __m128i m2)

(V)PSUBSW: __m128i _mm_subs_epi16(__m128i m1, __m128i m2)

VPSUBSB: __m256i _mm256_subs_epi8(__m256i m1, __m256i m2)

VPSUBSW: __m256i _mm256_subs_epi16(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PSUBUSB/PSUBUSW — Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F D8 /r PSUBUSB xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned byte integers in xmm2/m128 from packed unsigned byte integers in xmm1 and saturate result.
66 0F D9 /r PSUBUSW xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed unsigned word integers in xmm2/m128 from packed unsigned word integers in xmm1 and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2 and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2 and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2 and saturate result.

Instruction Operand Encoding

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

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: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (255:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (127:128) of the corresponding YMM register are zeroed.

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

Operation

VPSUBUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
```

VPSUBUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[VLMAX:128] ← 0
```

PSUBUSB (128-bit Legacy SSE Version)

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

DEST[VLMAX:128] (Unmodified)

VPSUBUSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);

VPSUBUSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[VLMAX:128] ← 0

PSUBUSW (128-bit Legacy SSE Version)
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSUBUSB: __m128i _mm_subs_epu8(__m128i m1, __m128i m2)

(V)PSUBUSW: __m128i _mm_subs_epu16(__m128i m1, __m128i m2)

VPSUBUSB: __m256i _mm256_subs_epu8(__m256i m1, __m256i m2)

VPSUBUSW: __m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ — Unpack High Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 68/r PUNPCKHBW xmm1,xmm2/m128	A	V/V	SSE2	Interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.
66 0F 69/r PUNPCKHWD xmm1,xmm2/m128	A	V/V	SSE2	Interleave high-order words from xmm1 and xmm2/m128 into xmm1.
66 0F 6A/r PUNPCKHDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave high-order double-words from xmm1 and xmm2/m128 into xmm1.
66 0F 6D/r PUNPCKHQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave high-order quad-word from xmm1 and xmm2/m128 into xmm1 register.
VEX.NDS.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6A /r VPUNPCKHDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order double-words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6D /r VPUNPCKHQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave high-order quad-word from xmm2 and xmm3/m128 into xmm1 register.
VEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 6A /r VPUNPCKHDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order double-words from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 6D /r VPUNPCKHQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave high-order quad-word from ymm2 and ymm3/m256 into ymm1 register.

Instruction Operand Encoding

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

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure F-2 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

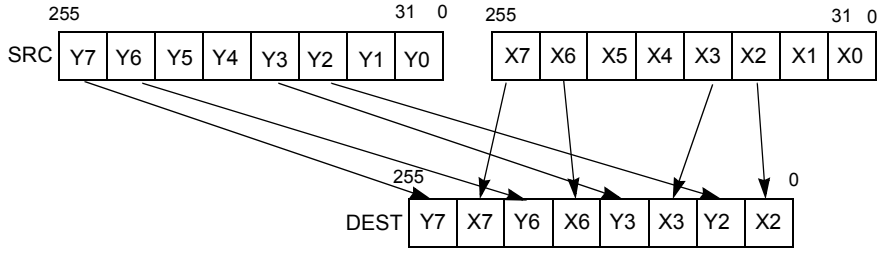


Figure E-1. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 128-bit memory operand an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

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

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (127:128) of the corresponding YMM register are zeroed.

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

Operation

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)

```
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ←SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
DEST[167:160] ← SRC1[215:208]
```

DEST[175:168] ← SRC2[215:208]
DEST[183:176] ← SRC1[223:216]
DEST[191:184] ←SRC2[223:216]
DEST[199:192] ← SRC1[231:224]
DEST[207:200] ← SRC2[231:224]
DEST[215:208] ← SRC1[239:232]
DEST[223:216] ← SRC2[239:232]
DEST[231:224] ← SRC1[247:240]
DEST[239:232] ← SRC2[247:240]
DEST[247:240] ← SRC1[255:248]
DEST[255:248] ← SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ←SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]
DEST[143:128] ← SRC1[207:192]
DEST[159:144] ← SRC2[207:192]
DEST[175:160] ← SRC1[223:208]
DEST[191:176] ← SRC2[223:208]

DEST[207:192] ← SRC1[239:224]
DEST[223:208] ← SRC2[239:224]
DEST[239:224] ← SRC1[255:240]
DEST[255:240] ← SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ←SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128] ← SRC1[255:192]
DEST[255:192] ← SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW (128-bit Legacy SSE Version)
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(DEST, SRC)

DEST[255:127] (Unmodified)

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

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

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

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

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

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

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

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

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)PUNPCKHBW: __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)

VPUNPCKHBW: __m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)

(V)PUNPCKHWD: __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)

VPUNPCKHWD: __m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)

(V)PUNPCKHDQ: __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)

VPUNPCKHDQ: __m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)

(V)PUNPCKHQDQ: __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)

VPUNPCKHQDQ: __m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ — Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 60/r PUNPCKLBW xmm1,xmm2/m128	A	V/V	SSE2	Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.
66 0F 61/r PUNPCKLWD xmm1,xmm2/m128	A	V/V	SSE2	Interleave low-order words from xmm1 and xmm2/m128 into xmm1.
66 0F 62/r PUNPCKLDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order double-words from xmm1 and xmm2/m128 into xmm1.
66 0F 6C/r PUNPCKLQDQ xmm1, xmm2/m128	A	V/V	SSE2	Interleave low-order quad-word from xmm1 and xmm2/m128 into xmm1 register.
VEX.NDS.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 62 /r VPUNPCKLDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order double-words from xmm2 and xmm3/m128 into xmm1.
VEX.NDS.128.66.0F.WIG 6C /r VPUNPCKLQDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Interleave low-order quad-word from xmm2 and xmm3/m128 into xmm1 register.
VEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 62 /r VPUNPCKLDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order double-words from ymm2 and ymm3/m256 into ymm1 register.
VEX.NDS.256.66.0F.WIG 6C /r VPUNPCKLQDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Interleave low-order quad-word from ymm2 and ymm3/m256 into ymm1 register.

Instruction Operand Encoding

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

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the first source operand and second source operand into the destination operand. (Figure 5-5 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

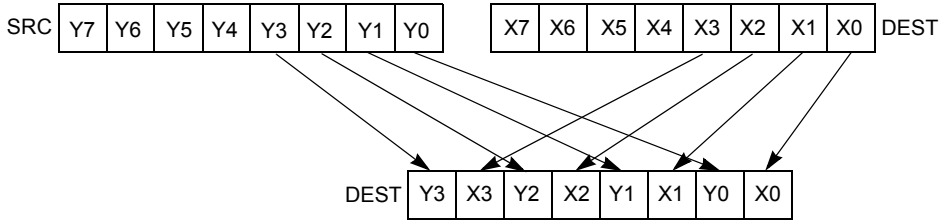


Figure 5-5. 128-bit PUNPCKLBW Instruction Operation using 64-bit Operands

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.

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

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

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (127:128) of the corresponding YMM register are zeroed.

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

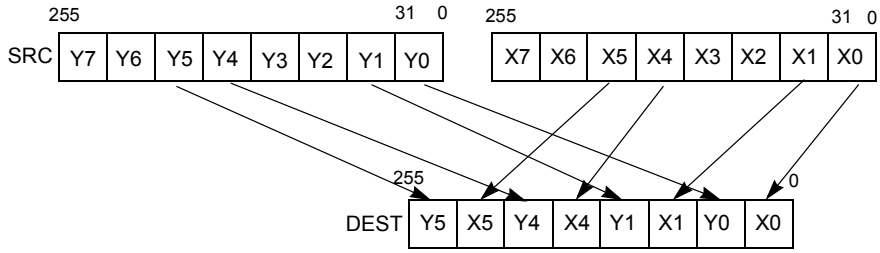


Figure E-2. 256-bit VPUNPCKLDQ Instruction Operation

Operation

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC1[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ←SRC2[31:24]

DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
DEST[135:128] ← SRC1[135:128]
DEST[143:136] ← SRC2[135:128]
DEST[151:144] ← SRC1[143:136]
DEST[159:152] ← SRC2[143:136]
DEST[167:160] ← SRC1[151:144]
DEST[175:168] ← SRC2[151:144]
DEST[183:176] ← SRC1[159:152]
DEST[191:184] ←SRC2[159:152]
DEST[199:192] ← SRC1[167:160]
DEST[207:200] ← SRC2[167:160]
DEST[215:208] ← SRC1[175:168]
DEST[223:216] ← SRC2[175:168]
DEST[231:224] ← SRC1[183:176]
DEST[239:232] ← SRC2[183:176]
DEST[247:240] ← SRC1[191:184]
DEST[255:248] ← SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ←SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
DEST[143:128] ← SRC1[143:128]
DEST[159:144] ← SRC2[143:128]
DEST[175:160] ← SRC1[159:144]
DEST[191:176] ← SRC2[159:144]
DEST[207:192] ← SRC1[175:160]
DEST[223:208] ← SRC2[175:160]
DEST[239:224] ← SRC1[191:176]
DEST[255:240] ← SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

PUNPCKLBW
DEST[127:0] ← INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_BYTES(SRC1, SRC2)
DEST[255:127] ← 0

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

PUNPCKLWD
DEST[127:0] ← INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLWD (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_WORDS(SRC1, SRC2)
DEST[255:127] ← 0

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

PUNPCKLDQ
DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLDQ (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[255:127] ← 0

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

PUNPCKLQDQ
DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLQDQ (VEX.128 encoded instruction)
DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[255:127] ← 0

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)PUNPCKLBW: __m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)

VPUNPCKLBW: __m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)

(V)PUNPCKLWD: __m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)

VPUNPCKLWD: __m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)

(V)PUNPCKLDQ: __m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)

VPUNPCKLDQ: __m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)

(V)PUNPCKLQDQ: __m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)

VPUNPCKLQDQ: __m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

PXOR — Exclusive Or

Opcode/ Instruction	Op/ En	64/3 2-bit Mode	CPUID Feature Flag	Description
66 0F EF /r PXOR xmm1, xmm2/m128	A	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.

Instruction Operand Encoding

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

Description

Performs a bitwise logical XOR operation on the second source operand and the first source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands differ, otherwise it is set to 0.

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

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

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (127:128) of the corresponding YMM register are zeroed.

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

Operation

VPXOR (VEX.256 encoded version)

DEST ← SRC1 XOR SRC2

VPXOR (VEX.128 encoded version)
DEST ← SRC1 XOR SRC2
DEST[VLMAX:128] ← 0

PXOR (128-bit Legacy SSE version)
DEST ← DEST XOR SRC
DEST[VLMAX:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PXOR: __m128i _mm_xor_si128 (__m128i a, __m128i b)

VPXOR: __m256i _mm256_xor_si256 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

MOVNTDQA — Load Double Quadword Non-Temporal Aligned Hint

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

Instruction Operand Encoding[1]

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

Description

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

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

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

1. ModRM.MOD = 011B required

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.

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

Operation

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

VMOVNTDQA (VEX.128 encoded form)
DEST ← SRC
DEST[VLMAX:128] ← 0

VMOVNTDQA (VEX.256 encoded form)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVNTDQA: __m128i _mm_stream_load_si128 (__m128i *p);

VMOVNTDQA: __m256i _mm256_stream_load_si256 (const __m256i *p);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type1; additionally

#UD	If VEX.vvvv != 1111B.

VBROADCAST — Broadcast Floating-Point Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	A	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VEX.128.66.0F38.W0 18/r VBROADCASTSS xmm1, xmm2	A	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, xmm2	A	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, xmm2	A	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.

Instruction Operand Encoding

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

Description

Take the low floating-point data element 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 an XMM register, only the low 32-bit or 64-bit data element is used.

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

An attempt to execute VBROADCASTSD encoded with VEX.L= 0 will cause an #UD exception.

Operation

VBROADCASTSS (128 bit version)
temp ← SRC[31:0]
FOR j ← 0 TO 3
DEST[31+j*32: j*32] ← temp
ENDFOR
DEST[VLMAX:128] ← 0

VBROADCASTSS (VEX.256 encoded version)
temp ← SRC[31:0]
FOR j ← 0 TO 7
DEST[31+j*32: j*32] ← temp
ENDFOR

VBROADCASTSD (VEX.256 encoded version)
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTSS: __m128 _mm_broadcastss_ps(__m128);

VBROADCASTSS: __m256 _mm256_broadcastss_ps(__m128);

VBROADCASTSD: __m256d _mm256_broadcastsd_pd(__m128d);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0 for VBROADCASTSD,
	If VEX.W = 1.

VBROADCASTF128/I128 — Broadcast 128-Bit Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	A	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	A	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.

Instruction Operand Encoding

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

Description

VBROADCASTF128 and VBROADCASTI128 load 128-bit data from the source operand (second operand) and broadcast to the destination operand (first operand).

The destination operand is a YMM register. The source operand is 128-bit memory location. Register source encodings for VBROADCASTF128 and VBROADCASTI128 are reserved and will #UD.

VBROADCASTF128 and VBROADCASTI128 are only supported as 256-bit wide versions.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. Attempts to execute any VPBROADCAST* instruction with VEX.W = 1 will cause #UD.

An attempt to execute VBROADCASTF128 or VBROADCASTI128 encoded with VEX.L= 0 will cause an #UD exception.

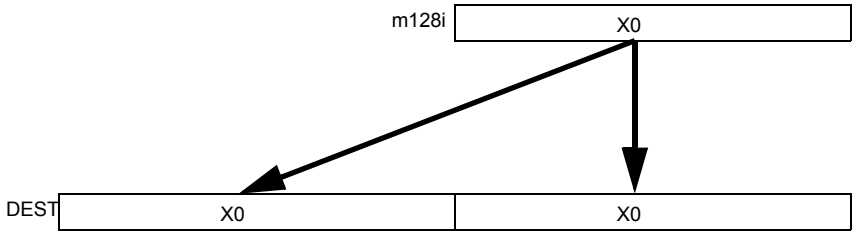


Figure 5-6. VBROADCASTI128 Operation

Operation

VBROADCASTF128/VBROADCASTI128

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
```

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTF128: __m256 _mm256_broadcast_ps(__m128 *);

VBROADCASTF128: __m256d _mm256_broadcast_pd(__m128d *);

VBROADCASTI128: __m256i _mm256_broadcastsi128_si256(__m128i);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0 ,
	If VEX.W = 1.

VPBLENDD — Blend Packed Dwords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLENDD xmm1, xmm2, xmm3/m128, imm8	A	V/V	AVX2	Select dwords from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.
VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLENDD ymm1, ymm2, ymm3/m256, imm8	A	V/V	AVX2	Select dwords from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.

Instruction Operand Encoding

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

Description

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

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

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

Operation

VPBLENDD (VEX.256 encoded version)

```
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
```

ELSE DEST[127:96] ← SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] ← SRC2[159:128]
ELSE DEST[159:128] ← SRC1[159:128]
IF (imm8[5] == 1) THEN DEST[191:160] ← SRC2[191:160]
ELSE DEST[191:160] ← SRC1[191:160]
IF (imm8[6] == 1) THEN DEST[223:192] ← SRC2[223:192]
ELSE DEST[223:192] ← SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] ← SRC2[255:224]
ELSE DEST[255:224] ← SRC1[255:224]

VPBLENDD (VEX.128 encoded version)
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[VLMAX:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDD: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)

VPBLENDD: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

VPBROADCAST — Broadcast Integer Data

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in xmm1.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in xmm1.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in mem to two locations in xmm1.
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in mem to four locations in ymm1.

Instruction Operand Encoding

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

Description

Load integer data from the source operand (second operand) and broadcast to all elements of the destination operand (first operand).

The destination operand is a YMM register. The source operand is 8-bit, 16-bit 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. VPBROADCASTB/D/W/Q also support XMM register as the source operand.

VPBROADCASTB/W/D/Q 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. Attempts to execute any VPBROADCAST* instruction with VEX.W = 1 will cause #UD.

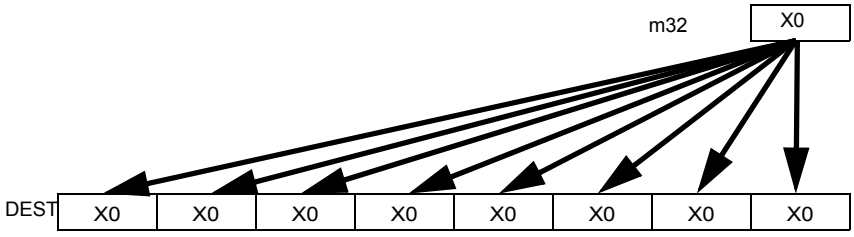


Figure 5-7. VPBROADCASTD Operation (VEX.256 encoded version)

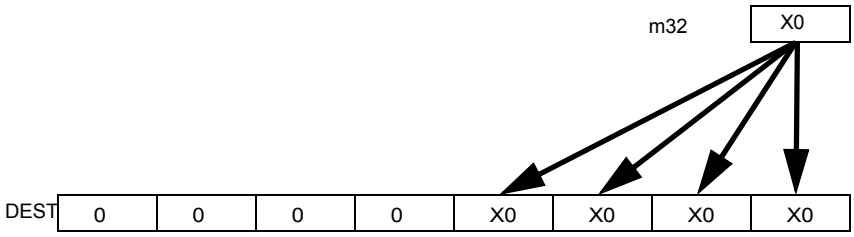
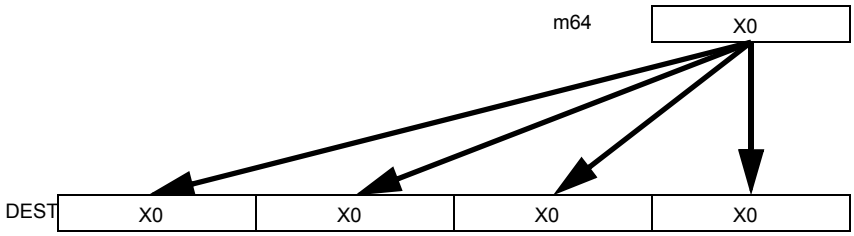


Figure 5-8. VPBROADCASTD Operation (128-bit version)



Operation

VPBROADCASTB (VEX.128 encoded version)
temp ← SRC[7:0]
FOR j ← 0 TO 15
DEST[7+j*8: j*8] ← temp
ENDFOR
DEST[VLMAX:128] ← 0

VPBROADCASTB (VEX.256 encoded version)
temp ← SRC[7:0]
FOR j ← 0 TO 31
DEST[7+j*8: j*8] ← temp
ENDFOR

VPBROADCASTW (VEX.128 encoded version)
temp ← SRC[15:0]
FOR j ← 0 TO 7
DEST[15+j*16: j*16] ← temp
ENDFOR
DEST[VLMAX:128] ← 0

VPBROADCASTW (VEX.256 encoded version)
temp ← SRC[15:0]
FOR j ← 0 TO 15
DEST[15+j*16: j*16] ← temp
ENDFOR

VPBROADCASTD (128 bit version)
temp ← SRC[31:0]
FOR j ← 0 TO 3
DEST[31+j*32: j*32] ← temp
ENDFOR
DEST[VLMAX:128] ← 0

VPBROADCASTD (VEX.256 encoded version)
temp ← SRC[31:0]
FOR j ← 0 TO 7
DEST[31+j*32: j*32] ← temp
ENDFOR

VPBROADCASTQ (VEX.128 encoded version)
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[VLMAX:128] ← 0

VPBROADCASTQ (VEX.256 encoded version)
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB: __m256i _mm256_broadcastb_epi8(__m128i);

VPBROADCASTW: __m256i _mm256_broadcastw_epi16(__m128i);

VPBROADCASTD: __m256i _mm256_broadcastd_epi32(__m128i);

VPBROADCASTQ: __m256i _mm256_broadcastq_epi64(__m128i);

VPBROADCASTB: __m128i _mm_broadcastb_epi8(__m128i);

VPBROADCASTW: __m128i _mm_broadcastw_epi16(__m128i);

VPBROADCASTD: __m128i _mm_broadcastd_epi32(__m128i);

VPBROADCASTQ: __m128i _mm_broadcastq_epi64(__m128i);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.W = 1.

VPERMD — Full Doublewords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute doublewords in ymm3/m256 using indexes in ymm2 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Use the index values in each dword element of the first source operand (the second operand) to select a dword element in the second source operand (the third operand), the resultant dword value from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMD encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMD: __m256i _mm256_permutevar8x32_epi32(__m256i a, __m256i offsets);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0 for VPERMD, If VEX.W = 1.

VPERMPD — Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute double-precision floating-point elements in ymm2/m256 using indexes in imm8 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Use two-bit index values in the immediate byte to select a double-precision floating-point element in the source operand; the resultant data from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple location in the destination operand.

An attempt to execute VPERMPD encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (VEX.256 encoded version)

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD: __m256d _mm256_permute4x64_pd(__m256d a, int control) ;

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0.

VPERMPS — Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 16 /r VPERMPS ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute single-precision floating-point elements in ymm3/m256 using indexes in ymm2 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Use the index values in each dword element of the first source operand (the second operand) to select a single-precision floating-point element in the second source operand (the third operand), the resultant data from the second source operand is copied to the destination operand (the first operand) in the corresponding position of the index element. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

An attempt to execute VPERMPS encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPS (VEX.256 encoded version)

```
DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS: __m256i _mm256_permutevar8x32_ps(__m256 a, __m256i offsets)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VPERMQ — Qwords Element Permutation

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute qwords in ymm2/m256 using indexes in imm8 and store the result in ymm1.

Instruction Operand Encoding

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

Description

Use two-bit index values in the immediate byte to select a qword element in the source operand, the resultant qword value from the source operand is copied to the corresponding element of the destination operand in the order of the index field. Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

An attempt to execute VPERMQ encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMQ (VEX.256 encoded version)

```
DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMQ: __m256i _mm256_permute4x64_epi64(__m256i a, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 0.

VPERM2I128 — Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 46 /r ib VPERM2I128 ymm1, ymm2, ymm3/m256, imm8	A	V/V	AVX2	Permute 128-bit integer data in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.

Instruction Operand Encoding

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

Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

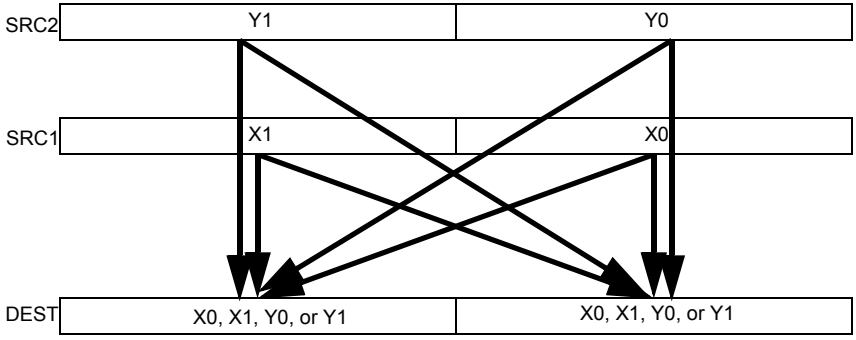


Figure 5-9. VPERM2I128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2I128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VEXTRACTI128 — Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Modet	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, imm8	A	V/V	AVX2	Extract 128 bits of integer data from ymm2 and store results in xmm1/mem.

Instruction Operand Encoding

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

Description

Extracts 128-bits of packed integer values from the source operand (second operand) at a 128-bit offset from imm8[0] into the destination operand (first operand). The destination may be either an XMM register or a 128-bit memory location.

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

The high 7 bits of the immediate are ignored.

An attempt to execute VEXTRACTI128 encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTI128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
```

VEXTRACTI128 (register destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[VLMAX:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTI128: __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	IF VEX.L = 0, If VEX.W = 1.

VINSERTI128 — Insert Packed Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX2	Insert 128-bits of integer data from xmm3/mem and the remaining values from ymm2 into ymm1

Instruction Operand Encoding

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

Description

Performs an insertion of 128-bits of packed integer data from the second source operand (third operand) into an the destination operand (first operand) at a 128-bit offset from imm8[0]. The remaining portions of the destination are written by the corresponding fields of the first source operand (second operand). The second source operand can be either an XMM register or a 128-bit memory location.

The high 7 bits of the immediate are ignored.

VEX.L must be 1; an attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI128

```
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ←SRC2[127:0]
    1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ←TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTI128: __m256i _mm256_inserti128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOVD xmm1, xmm2, m128	A	V/V	AVX2	Conditionally load dword values from m128 using mask in xmm2 and store in xmm1
VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD ymm1, ymm2, m256	A	V/V	AVX2	Conditionally load dword values from m256 using mask in ymm2 and store in ymm1
VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ xmm1, xmm2, m128	A	V/V	AVX2	Conditionally load qword values from m128 using mask in xmm2 and store in xmm1
VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVQ ymm1, ymm2, m256	A	V/V	AVX2	Conditionally load qword values from m256 using mask in ymm2 and store in ymm1
VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD m128, xmm1, xmm2	B	V/V	AVX2	Conditionally store dword values from xmm2 using mask in xmm1
VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVD m256, ymm1, ymm2	B	V/V	AVX2	Conditionally store dword values from ymm2 using mask in ymm1
VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ m128, xmm1, xmm2	B	V/V	AVX2	Conditionally store qword values from xmm2 using mask in xmm1
VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVQ m256, ymm1, ymm2	B	V/V	AVX2	Conditionally store qword values from ymm2 using mask in ymm1

Instruction Operand Encoding

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

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VPMASKMOVD - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
```

DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0

VPMASKMOVD -128-bit load
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX:128] ← 0

VPMASKMOVQ - 256-bit load
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0

VPMASKMOVQ - 128-bit load
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX:128] ← 0

VPMASKMOVD - 256-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ←SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]

VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VPMASKMOVQ - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]
IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

VPMASKMOVQ - 128-bit store

```
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)

VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)

VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);

VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);

VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)

VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)

VPMASKMOVQ: __m128i _mm_maskload_epi64(__int cont *a, __m128i mask);

VPMASKMOVQ: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations).

VPSLLVD/VPSLLVQ — Variable Bit Shift Left Logical

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

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

Operation

VPSLLVD (VEX.128 version)

```
COUNT_0 ← SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX:128] ← 0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ← 0;
```

VPSLLVQ (VEX.128 version)

```
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
```

```
IF COUNT_1 < 64 THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX:128] ← 0;
```

VPSLLVQ (VEX.256 version)

```
COUNT_0 ← SRC2[5 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVD: __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)

VPSLLVD: __m128i _mm_sllv_epi32 (__m128i m, __m128i count)

VPSLLVQ: __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

VPSLLVQ: __m128i _mm_sllv_epi64 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VPSRAVD — Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in the sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in the sign bits.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual doubleword data elements in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in each data element are shifted right, the empty high-order bits are filled with the sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31, then the destination data element are filled with the corresponding sign bit of the source element.

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

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

Operation

VPSRAVD (VEX.128 version)

```
COUNT_0 ← SRC2[31: 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 112];
IF COUNT_0 < 32 THEN
```

```
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31] );
FI;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
ELSE
    For (i = 0 to 31) DEST[i + 96] ← (SRC1[127] );
FI;
DEST[VLMAX:128] ← 0;
```

VPSRAVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
ELSE
    For (i = 0 to 31) DEST[i + 0] ← (SRC1[31] );
FI;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
ELSE
    For (i = 0 to 31) DEST[i + 224] ← (SRC1[255] );
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD: __m256i _mm256_srav_epi32 (__m256i m, __m256i count)

VPSRAVD: __m128i _mm_srav_epi32 (__m128i m, __m128i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

VPSRLVD/VPSRLVQ — Variable Bit Shift Right Logical

Opcode/ Instruction	Op/ EN	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.

Instruction Operand Encoding

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

Description

Shifts the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

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

Operation

VPSRLVD (VEX.128 version)

```
COUNT_0 ← SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[VLMAX:128] ← 0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
DEST[255:224] ← 0;
```

VPSRLVQ (VEX.128 version)

```
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
```

```
IF COUNT_1 < 64THEN
DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
DEST[127:64] ← 0;
DEST[VLMAX:128] ← 0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 ← SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVD: __m256i _mm256_srlv_epi32 (__m256i m, __m256i count);

VPSRLVD: __m128i _mm_srlv_epi32 (__m128i m, __m128i count);

VPSRLVQ: __m256i _mm256_srlv_epi64 (__m256i m, __m256i count);

VPSRLVQ: __m128i _mm_srlv_epi64 (__m128i m, __m128i count);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4

VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/3 2-bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD xmm1, vm32x, xmm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD xmm1, vm64x, xmm2	A	V/V	AVX2	Using qword indices specified in vm64x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD ymm1, vm32x, ymm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD ymm1, vm64y, ymm2	A	V/V	AVX2	Using qword indices specified in vm64y, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VGATHERDPD (VEX.128 version)
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
```

```
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[VLMAX:128] ← 0;
DEST[VLMAX:128] ← 0;
```
(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPD (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[VLMAX:128] ← 0;
DEST[VLMAX:128] ← 0;
```
(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPD (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
```

```
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERDPD (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);

VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);

VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);

VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS xmm1, vm32x, xmm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64x, xmm2	A	V/V	AVX2	Using qword indices specified in vm64x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS ymm1, vm32y, ymm2	A	V/V	AVX2	Using dword indices specified in vm32y, gather single-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64y, xmm2	A	V/V	AVX2	Using qword indices specified in vm64y, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VGATHERDPS (VEX.128 version)
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX:128] ← 0;
```

```
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[VLMAX:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERQPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX:64] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERDPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
```

```
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

VGATHERQPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX:128] ← 0;
DEST[VLMAX:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);

VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);

VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);

VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);

VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERQPS: __m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);

VGATHERQPS: __m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD xmm1, vm32x, xmm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64x, xmm2	A	V/V	AVX2	Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD ymm1, vm32y, ymm2	A	V/V	AVX2	Using dword indices specified in vm32y, gather dword from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64y, xmm2	A	V/V	AVX2	Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 dword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDD (VEX.128 version)
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX:128] ← 0;
```

```
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[VLMAX:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERQD (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[VLMAX:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX:64] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERDD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
```

```
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERQD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← 0xFFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX:128] ← 0;
DEST[VLMAX:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDD: __m256i _mm256_i32gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);

VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ xmm1, vm32x, xmm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ xmm1, vm64x, xmm2	A	V/V	AVX2	Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ ymm1, vm32x, ymm2	A	V/V	AVX2	Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ ymm1, vm64y, ymm2	A	V/V	AVX2	Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose

register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable -

given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VPGATHERDQ (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
MASK[VLMAX:128] ← 0;
DEST[VLMAX:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERQQ (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    i ←j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
MASK[VLMAX:128] ← 0;
DEST[VLMAX:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDQ (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
```

```
    IF MASK[63+i] THEN
        MASK[i +63:i] ← 0xFFFFFFFF_FFFFFFFF; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDQ: __m128i _mm_i32gather_epi64 (int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDQ: __m256i _mm256_i32gather_epi64 (int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, int64 const * base, __m128i index, __m256i mask, const int scale);

VPGATHERQQ: __m128i _mm_i64gather_epi64 (int64 const * base, __m128i index, const int scale);

VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQQ: __m256i _mm256_i64gather_epi64 (int64 const * base, __m256i index, const int scale);

VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, int64 const * base, __m256i index, __m256i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12

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.

VFMADD132PD/VFMADD213PD/VFMADD231PD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 98 /r VFMADD132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 98 /r VFMADD132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results

in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD213PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD231PD: __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);

VFMADD132PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD213PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

VFMADD231PD: __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132PS/VFMADD213PS/VFMADD231PS - Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 98 /r VFMADD132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 98 /r VFMADD132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 B8 /r VFMADD231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed

by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
```

```
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PS: __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD213PS: __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD231PS: __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);

VFMADD132PS: __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

VFMADD213PS: __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

VFMADD231PS: __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADD132SD/VFMADD213SD/VFMADD231SD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 99 /r VFMADD132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 A9 /r VFMADD213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 B9 /r VFMADD231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-add computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFMADD213SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFMADD231SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SD: __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

VFMADD213SD: __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

VFMADD231SD: __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 99 /r VFMADD132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 A9 /r VFMADD213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, add to xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 B9 /r VFMADD231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior"

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADD132SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMADD213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMADD231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[63:0] + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SS: __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

VFMADD213SS: __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

VFMADD231SS: __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFMADDSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192]
FI
```

VFMADDSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192]
FI
```

VFMADDSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB213PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB231PD: __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);

VFMADDSUB132PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB213PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

VFMADDSUB231PD: __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFMADDSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] +
SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] +
SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] +
DEST[n+63:n+32])
}
```

```
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUB132PS: __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB213PS: __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB231PS: __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);

VFMADDSUB132PS: __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

VFMADDSUB213PS: __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

VFMADDSUB231PS: __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFMSUBADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192]
FI
```

VFMSUBADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192]
FI
```

VFMSUBADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[VLMAX-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD213PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD231PD: __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);

VFMSUBADD132PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD213PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

VFMSUBADD231PD: __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS - Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFMSUBADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -
SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -
SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
```

```
For i = 0 to MAXVL -1{
    n = 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -
DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PS: __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD213PS: __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD231PS: __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);

VFMSUBADD132PS: __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

VFMSUBADD213PS: __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

VFMSUBADD231PS: __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source

operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PD: __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB213PD: __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB231PD: __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);

VFMSUB132PD: __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

VFMSUB213PD: __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

VFMSUB231PD: __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 BA /r VFMSUB231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source

operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PS: __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB213PS: __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB231PS: __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);

VFMSUB132PS: __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

VFMSUB213PS: __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

VFMSUB231PS: __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9B /r VFMSUB132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 AB /r VFMSUB213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 BB /r VFMSUB231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFMSUB213SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFMSUB231SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SD: __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

VFMSUB213SD: __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

VFMSUB231SD: __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS - Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9B /r VFMSUB132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 AB /r VFMSUB213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, subtract xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 BB /r VFMSUB231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand,

performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMSUB213SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

VFMSUB231SS DEST, SRC2, SRC3

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132SS: __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

VFMSUB213SS: __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

VFMSUB231SS: __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, negate the multiplication result and add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFNMADD132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
```

```
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PD: __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD213PD: __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD231PD: __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);

VFNMADD132PD: __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

VFNMADD213PD: __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

VFNMADD231PD: __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, negate the multiplication result and add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 BC /r VFNMADD231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFNMADD132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
```

```
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD213PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD231PS: __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);

VFNMADD132PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD213PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

VFNMADD231PS: __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9D /r VFNMADD132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 AD /r VFNMADD213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 BD /r VFNMADD231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFNMADD213SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFNMADD231SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132SD: __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

VFNMADD213SD: __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

VFNMADD231SD: __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS - Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9D /r VFNMADD132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 AD /r VFNMADD213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and add to xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 BD /r VFNMADD231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0
```

VFNMADD213SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0
```

VFNMADD231SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADD132SS: __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

VFNMADD213SS: __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

VFNMADD231SS: __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm0 and ymm1, negate the multiplication result and subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFNMSUB132PD: Multiplies the two or four packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two or four packed double-precision floating-point values from the second source to the two or four packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =2
ELSEIF (VEX.256)
    MAXVL = 4
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
```

```
IF (VEX.128) THEN
DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB213PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB231PD: __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);

VFNMSUB132PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB213PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

VFNMSUB231PD: __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS - Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm0, xmm1, xmm2/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm0 and ymm1, negate the multiplication result and subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0F38.0 BE /r VFNMSUB231PS ymm0, ymm1, ymm2/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0 and put result in ymm0.

Instruction Operand Encoding

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

Description

VFNMSUB132PS: Multiplies the four or eight packed single-precision floating-point values from the first source operand to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four or eight packed single-precision floating-point values from the second source operand to the four or eight packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four or eight packed single-precision floating-point values from the second source to the four or eight packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four or eight packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four or eight packed single-precision floating-point values to the destination operand (first source operand).

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”.

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXVL =4
ELSEIF (VEX.256)
    MAXVL = 8
FI
For i = 0 to MAXVL-1 {
    n = 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
```

```
IF (VEX.128) THEN
    DEST[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB213PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB231PS: __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);

VFNMSUB132PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

VFNMSUB213PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

VFNMSUB231PS: __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W1 9F /r VFNMSUB132SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 AF /r VFNMSUB213SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W1 BF /r VFNMSUB231SD xmm0, xmm1, xmm2/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

VFNMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 64-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFNMSUB213SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

VFNMSUB231SD DEST, SRC2, SRC3

```
DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SD: __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

VFNMSUB213SD: __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

VFNMSUB231SD: __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS - Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.LIG.128.66.0F38.W0 9F /r VFNMSUB132SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1 and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 AF /r VFNMSUB213SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm0 and xmm1, negate the multiplication result and subtract xmm2/mem and put result in xmm0.
VEX.DDS.LIG.128.66.0F38.W0 BF /r VFNMSUB231SS xmm0, xmm1, xmm2/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0 and put result in xmm0.

Instruction Operand Encoding

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

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 32-bit memory location and encoded in rm_field. The upper bits ([255:128]) of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

Operation

In the operations below, "+", "-", and "*" symbols represent addition, subtraction, and multiplication operations with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0
```

VFNMSUB213SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0
```

VFNMSUB231SS DEST, SRC2, SRC3

```
DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUB132SS: __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

VFNMSUB213SS: __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

VFNMSUB231SS: __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 3

CHAPTER 7
INSTRUCTION SET REFERENCE - VEX-ENCODED GPR INSTRUCTIONS

This chapter describes the various general-purpose instructions, the majority of which are encoded using VEX prefix.

7.1 INSTRUCTION FORMAT

The format used for describing each instruction as in the example below is described in Chapter 5.

ANDN — Logical AND NOT (THIS IS AN EXAMPLE)

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32	A	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.NDS.LZ. 0F38.W1 F2 /r ANDN r64a, r64b, r/m64	A	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64b.

Instruction Operand Encoding

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

7.2 INSTRUCTION SET REFERENCE

ANDN — Logical AND NOT

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32	A	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.NDS.LZ. 0F38.W1 F2 /r ANDN r64a, r64b, r/m64	A	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64a.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
DEST ← (NOT SRC1) bitwiseAND SRC2;
SF ← DEST[OperandSize -1];
ZF ← (DEST = 0);
```

Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

BEXTR — Bit Field Extract

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS[1].LZ.0F38.W0 F7 /r BEXR r32a, r/m32, r32b	A	V/V	BMI1	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.NDS[1].LZ.0F38.W1 F7 /r BEXR r64a, r/m64, r64b	A	V/N.E.	BMI1	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a

NOTES:

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

Instruction Operand Encoding

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

Description

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the first source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize -1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
START ← SRC2[7:0];
LEN ← SRC2[15:8];
TEMP ← ZERO_EXTEND_TO_512 (SRC1 );
DEST ← ZERO_EXTEND(TEMP[START+LEN -1: START]);
ZF ← (DEST = 0);
```

Flags Affected

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

BEXTR: unsigned __int32 _bextr_u32(unsigned __int32 src, unsigned __int32 start. unsigned __int32 len);

BEXTR: unsigned __int64 _bextr_u64(unsigned __int64 src, unsigned __int32 start. unsigned __int32 len);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

BLSI — Extract Lowest Set Isolated Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /3 BLSI r32, r/m32	A	V/V	BMI1	Extract lowest set bit from r/m32 and set that bit in r32.
VEX.NDD.LZ.0F38.W1 F3 /3 BLSI r64, r/m64	A	V/N.E.	BMI1	Extract lowest set bit from r/m64, and set that bit in r64.

Instruction Operand Encoding

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

Description

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (-SRC) bitwiseAND (SRC);
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 0;
ELSE
    CF ← 1;
FI
DEST ← temp;
```

Flags Affected

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSI: unsigned __int32 _blsi_u32(unsigned __int32 src);

BLSI: unsigned __int64 _blsi_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

BLSMSK — Get Mask Up to Lowest Set Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /2 BLSMSK r32, r/m32	A	V/V	BMI1	Set all lower bits in r32 to "1" starting from bit 0 to lowest set bit in r/m32.
VEX.NDD.LZ.0F38.W1 F3 /2 BLSMSK r64, r/m64	A	V/N.E.	BMI1	Set all lower bits in r64 to "1" starting from bit 0 to lowest set bit in r/m64.

Instruction Operand Encoding

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

Description

Sets all the lower bits of the destination operand to "1" up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) XOR (SRC) ;
SF ← temp[OperandSize -1];
ZF ← 0;
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

SF is updated based on the result. CF is set if the source if zero. ZF and OF flags are cleared. AF and PF flag are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSMSK: unsigned __int32 _blsmsk_u32(unsigned __int32 src);

BLSMSK: unsigned __int64 _blsmsk_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

BLSR — Reset Lowest Set Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /1 BLSR r32, r/m32	A	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.NDD.LZ.0F38.W1 F3 /1 BLSR r64, r/m64	A	V/N.E.	BMI1	Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.

Instruction Operand Encoding

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

Description

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) bitwiseAND ( SRC );
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSR: unsigned __int32 _blsr_u32(unsigned __int32 src);

BLSR: unsigned __int64 _blsr_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

BZHI — Zero High Bits Starting with Specified Bit Position

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS[1].LZ.0F38.W0 F5 /r BZHI r32a, r/m32, r32b	A	V/V	BMI2	Zero bits in r/m32 starting with the position in r32b, write result to r32a.
VEX.NDS[1].LZ.0F38.W1 F5 /r BZHI r64a, r/m64, r64b	A	V/N.E.	BMI2	Zero bits in r/m64 starting with the position in r64b, write result to r64a.

NOTES:

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

Instruction Operand Encoding

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

Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize -1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize -1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
N ← SRC2[7:0]
DEST ← SRC1
IF (N < OperandSize)
    DEST[OperandSize-1:N] ← 0
FI
IF (N > OperandSize - 1)
    CF ← 1
ELSE
    CF ← 0
```

FI

Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BZHI: unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);

BZHI: unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

LZCNT— Count the Number of Leading Zero Bits

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BD /r LZCNT r16, r/m16	A	V/V	LZCNT	Count the number of leading zero bits in r/m16, return result in r16.
F3 0F BD /r LZCNT r32, r/m32	A	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
REX.W + F3 0F BD /r LZCNT r64, r/m64	A	V/N.E.	LZCNT	Count the number of leading zero bits in r/m64, return result in r64.

Instruction Operand Encoding

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

Description

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR.

In 64-bit mode 64-bit operand size requires REX.W=1.

Operation

```
temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp ← temp - 1
    DEST ← DEST+ 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
```

```
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

Flags Affected

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

LZCNT: unsigned __int32 _lzcnt_u32(unsigned __int32 src);

LZCNT: unsigned __int64 _lzcnt_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-21.

MULX — Unsigned Multiply Without Affecting Flags

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.F2.0F38.W0 F6 /r MULX r32a, r32b, r/m32	A	V/V	BMI2	Unsigned multiply of r/m32 with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F38.W1 F6 /r MULX r64a, r64b, r/m64	A	V/N.E.	BMI2	Unsigned multiply of r/m64 with RDX without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
        DEST2 ← (SRC1*SRC2)[63:0];
```

```
        DEST1 ← (SRC1*SRC2)[127:64];
FI
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);

unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

PDEP — Parallel Bits Deposit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F2.0F38.W0 F5 /r PDEP r32a, r32b, r/m32	A	V/V	BMI2	Parallel deposit of bits from r32b using mask in r/m32, result is written to r32a.
VEX.NDS.LZ.F2.0F38.W1 F5 /r PDEP r64a, r64b, r/m64	A	V/N.E.	BMI2	Parallel deposit of bits from r64b using mask in r/m64, result is written to r64a

Instruction Operand Encoding

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

Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

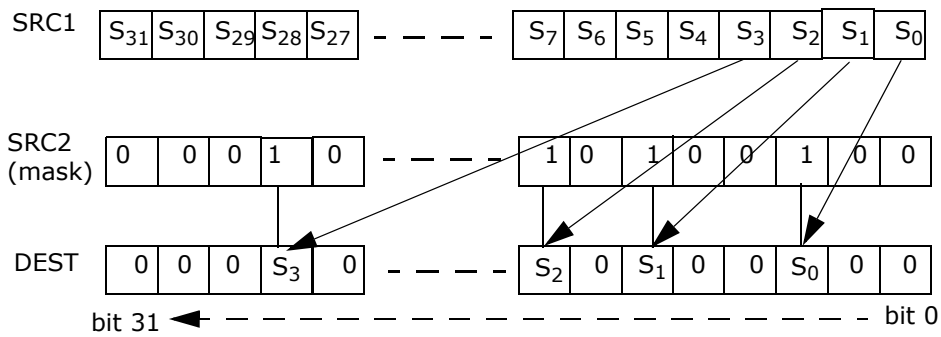


Figure 7-1. PDEP Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

        IF MASK[ m] = 1 THEN
            DEST[ m] ← TEMP[ k];
            k ← k+ 1;
        FI
        m ← m+ 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PDEP: unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);

PDEP: unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

PEXT — Parallel Bits Extract

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F5 /r PEXT r32a, r32b, r/m32	A	V/V	BMI2	Parallel extract of bits from r32b using mask in r/m32, result is written to r32a.
VEX.NDS.LZ.F3.0F38.W1 F5 /r PEXT r64a, r64b, r/m64	A	V/N.E.	BMI2	Parallel extract of bits from r64b using mask in r/m64, result is written to r64a.

Instruction Operand Encoding

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

Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

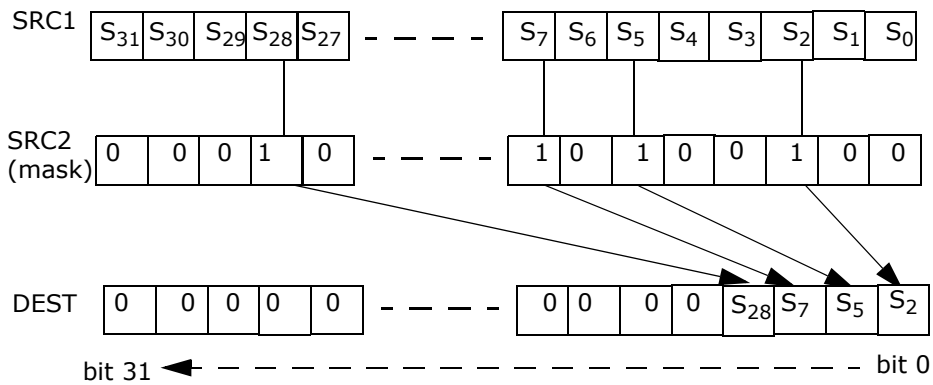


Figure 7-2. PEXT Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

        IF MASK[ m] = 1 THEN
            DEST[ k] ← TEMP[ m];
            k ← k+ 1;
        FI
        m ← m+ 1;

OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PEXT: unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);

PEXT: unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

RORX — Rotate Right Logical Without Affecting Flags

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, r/m32, imm8	A	V/V	BMI2	Rotate 32-bit r/m32 right imm8 times without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, r/m64, imm8	A	V/N.E.	BMI2	Rotate 64-bit r/m64 right imm8 times without affecting arithmetic flags.

Instruction Operand Encoding

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

Description

Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
IF (OperandSize = 32)
    y ← imm8 AND 1FH;
    DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64 )
    y ← imm8 AND 3FH;
    DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

SARX/SHLX/SHRX — Shift Without Affecting Flags

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS[1].LZ.F3.0F38.W0 F7 /r SARX r32a, r/m32, r32b	A	V/V	BMI2	Shift r/m32 arithmetically right with count specified in r32b.
VEX.NDS[1].LZ.66.0F38.W0 F7 /r SHLX r32a, r/m32, r32b	A	V/V	BMI2	Shift r/m32 logically left with count specified in r32b.
VEX.NDS[1].LZ.F2.0F38.W0 F7 /r SHRX r32a, r/m32, r32b	A	V/V	BMI2	Shift r/m32 logically right with count specified in r32b.
VEX.NDS[1].LZ.F3.0F38.W1 F7 /r SARX r64a, r/m64, r64b	A	V/N.E.	BMI2	Shift r/m64 arithmetically right with count specified in r64b.
VEX.NDS[1].LZ.66.0F38.W1 F7 /r SHLX r64a, r/m64, r64b	A	V/N.E.	BMI2	Shift r/m64 logically left with count specified in r64b.
VEX.NDS[1].LZ.F2.0F38.W1 F7 /r SHRX r64a, r/m64, r64b	A	V/N.E.	BMI2	Shift r/m64 logically right with count specified in r64b.

NOTES:

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

Instruction Operand Encoding

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

Description

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize -1, the COUNT value is masked.

SARX,SHRX, and SHLX instructions do not update flags.

Operation

```
TEMP ← SRC1;
IF VEX.W1 and CS.L = 1
THEN
    countMASK ←3FH;
ELSE
    countMASK ←1FH;
FI
COUNT ← (SRC2 AND countMASK)

DEST[OperandSize -1] = TEMP[OperandSize -1];
DO WHILE (COUNT != 0)
    IF instruction is SHLX
        THEN
            DEST[] ← DEST *2;
        ELSE IF instruction is SHRX
            THEN
                DEST[] ← DEST /2; //unsigned divide
        ELSE            // SARX
                DEST[] ← DEST /2; // signed divide, round toward negative infinity
    FI;
    COUNT ← COUNT - 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-22.

TZCNT — Count the Number of Trailing Zero Bits

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BC /r TZCNT r16, r/m16	A	V/V	BMI1	Count the number of trailing zero bits in r/m16, return result in r16.
F3 0F BC /r TZCNT r32, r/m32	A	V/V	BMI1	Count the number of trailing zero bits in r/m32, return result in r32
REX.W + F3 0F BC /r TZCNT r64, r/m64	A	V/N.E.	BMI1	Count the number of trailing zero bits in r/m64, return result in r64.

Instruction Operand Encoding

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

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

```
temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )

    temp ← temp +1
    DEST ← DEST+ 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
```

```
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

TZCNT: unsigned __int32 _tzcnt_u32(unsigned __int32 src);

TZCNT: unsigned __int64 _tzcnt_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-21.

INVPCID — Invalidate Processor Context ID

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
66 0F 38 82 /r INVPCID r32, m128	A	NE/V	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 0F 38 82 /r INVPCID r64, m128	A	V/NE	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

Instruction Operand Encoding

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

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on the invalidation type specified in the first operand and processor context identifier (PCID) invalidate descriptor specified in the second operand. The INVPCID descriptor is specified as a 16-byte memory operand and has no alignment restriction.

The layout of the INVPCID descriptor is shown in Figure 7-3. In 64-bit mode the linear address field (bits 127:64) in the INVPCID descriptor must satisfy canonical requirement unless the linear address field is ignored.

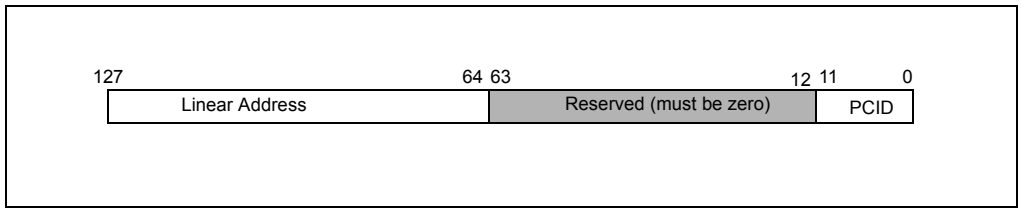


Figure 7-3. INVPCID Descriptor

Outside IA-32e mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits; however, if bits 63:32 of the register operand are not zero, INVPCID fails due to an attempt to use an unsupported INVPCID type (see below).

The INVPCID types supported by a logical processors are:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings for a single linear address and tagged with the PCID specified in the INVPCID descriptor, except global translations. The instruction may also invalidate global translations, mappings for other linear addresses, or mappings tagged with other PCIDs.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings tagged with the PCID specified in the INVPCID descriptor except global translations. In some cases, it may invalidate mappings for other PCIDs as well.
- All-context invalidation: If the INVPCID type is 2, the logical processor invalidates all mappings tagged with any PCID.
- All-context invalidation, retaining global translations: If the INVPCID type is 3, the logical processor invalidates all mappings tagged with any PCID except global translations, ignoring the INVPCID descriptor. The instruction may also invalidate global translations as well.

If an unsupported INVPCID type is specified, or if the reserved field in the descriptor is not zero, the instruction fails.

Outside IA-32e mode, the processor treats INVPCID as if all mappings are associated with PCID 000H.

Operation

```
INVPCID_TYPE ← value of register operand; // must be in the range of 0-3
INVPCID_DESC ← value of memory operand;
CASE INVPCID_TYPE OF
    0:              // individual-address invalidation retaining global translations
        OP_PCID ← INVPCID_DESC[11:0];
        ADDR ← INVPCID_DESC[127:64];
        Invalidate mappings for ADDR tagged with OP_PCID except global translations;
        BREAK;
    1:              // single PCID invalidation retaining globals
        OP_PCID ← INVPCID_DESC[11:0];
        Invalidate all mappings tagged with OP_PCID except global translations;
        BREAK;
    2:              // all PCID invalidation
        Invalidate all mappings tagged with any PCID;
        BREAK;
    3:              // all PCID invalidation retaining global translations
        Invalidate all mappings tagged with any PCID except global translations;
        BREAK;
ESAC;
```

Intel C/C++ Compiler Intrinsic Equivalent

INVPCID: void _invpcid(unsigned __int32 type, void * descriptor);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.
	If the source operand is located in an execute-only code segment.
	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If CR4.PCIDE=0, INVPCID_DESC[11:0] is not zero, and INVPCID_TYPE is either 0, or 1.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If an invalid type is specified in the register operand, i.e INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If CR4.PCIDE=0, INVPCID_DESC[11:0] is not zero, and INVPCID_TYPE is either 0, or 1.
#UD	If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
	If an invalid type is specified in the register operand.
	If an invalid type is specified in the register operand, i.e INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If CR4.PCIDE=0, INVPCID_DESC[11:0] is not zero, and INVPCID_TYPE is either 0, or 1.
	If INVPCID_TYPE is 0, INVPCID_DESC[127:64] is not a canonical address.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.

CHAPTER 8
INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

8.1 OVERVIEW

This chapter describes the software programming interface to the Intel® Transactional Synchronization Extensions of the Intel 64 architecture.

Multithreaded applications take advantage of increasing number of cores to achieve high performance. However, writing multi-threaded applications requires programmers to reason about data sharing among multiple threads. Access to shared data typically requires synchronization mechanisms. These mechanisms ensure multiple threads update shared data by serializing operations on the shared data, often through the use of a critical section protected by a lock. Since serialization limits concurrency, programmers try to limit synchronization overheads. They do this either through minimizing the use of synchronization or through the use of fine-grain locks; where multiple locks each protect different shared data. Unfortunately, this process is difficult and error prone; a missed or incorrect synchronization can cause an application to fail. Conservatively adding synchronization and using coarser granularity locks, where a few locks each protect many items of shared data, helps avoid correctness problems but limits performance due to excessive serialization. While programmers must use static information to determine when to serialize, the determination as to whether actually to serialize is best done dynamically.

8.2 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

Intel® Transactional Synchronization Extensions (Intel® TSX) allow the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to perform serialization only when required. This lets the processor to expose and exploit concurrency hidden in an application due to dynamically unnecessary synchronization.

With Intel TSX, programmer-specified code regions (also referred to as **transactional regions**) are executed transactionally. If the transactional execution completes successfully, then all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other logical processors. A processor makes architectural updates performed within the region visible to other logical processors only on a successful commit, a process referred to as an **atomic commit**.

Intel TSX provides two software interfaces to specify regions of code for transactional execution. **Hardware Lock Elision (HLE)** is a legacy compatible instruction set

extension (comprising the XACQUIRE and XRELEASE prefixes) to specify transactional regions. **Restricted Transactional Memory (RTM)** is a new instruction set interface (comprising the XBEGIN, XEND, and XABORT instructions) for programmers to define transactional regions in a more flexible manner than that possible with HLE. HLE is for programmers who prefer the backward compatibility of the conventional mutual exclusion programming model and would like to run HLE-enabled software on legacy hardware but would also like to take advantage of the new lock elision capabilities on hardware with HLE support. RTM is for programmers who prefer a flexible interface to the transactional execution hardware. In addition, Intel TSX also provides an XTEST instruction. This instruction allows software to query whether the logical processor is transactionally executing in a transactional region identified by either HLE or RTM.

Since a successful transactional execution ensures an atomic commit, the processor executes the code region optimistically without explicit synchronization. If synchronization was unnecessary for that specific execution, execution can commit without any cross-thread serialization. If the processor cannot commit atomically, the optimistic execution fails. When this happens, the processor will roll back the execution, a process referred to as a **transactional abort**. On a transactional abort, the processor will discard all updates performed in the region, restore architectural state to appear as if the optimistic execution never occurred, and resume execution non-transactionally.

A processor can perform a transactional abort for numerous reasons. A primary cause is due to conflicting accesses between the transactionally executing logical processor and another logical processor. Such conflicting accesses may prevent a successful transactional execution. Memory addresses read from within a transactional region constitute the **read-set** of the transactional region and addresses written to within the transactional region constitute the **write-set** of the transactional region. Intel TSX maintains the read- and write-sets at the granularity of a cache line. A conflicting access occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. A conflicting access typically means serialization is indeed required for this code region. Since Intel TSX detects data conflicts at the granularity of a cache line, unrelated data locations placed in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited transactional resources. For example, the amount of data accessed in the region may exceed an implementation-specific capacity. Additionally, some instructions and system events may cause transactional aborts. Frequent transactional aborts cause wasted cycles.

8.2.1 Hardware Lock Elision

Hardware Lock Elision (HLE) provides a legacy compatible instruction set interface for programmers to do transactional execution. HLE provides two new instruction prefix hints: XACQUIRE and XRELEASE.

The programmer uses the XACQUIRE prefix in front of the instruction that is used to acquire the lock that is protecting the critical section. The processor treats the indication as a hint to elide the write associated with the lock acquire operation. Even though the lock acquire has an associated write operation to the lock, the processor does not add the address of the lock to the transactional region's write-set nor does it issue any write requests to the lock. Instead, the address of the lock is added to the read-set. The logical processor enters transactional execution. If the lock was available before the XACQUIRE prefixed instruction, all other processors will continue to see it as available afterwards. Since the transactionally executing logical processor neither added the address of the lock to its write-set nor performed externally visible write operations to it, other logical processors can read the lock without causing a data conflict. This allows other logical processors to also enter and concurrently execute the critical section protected by the lock. The processor automatically detects any data conflicts that occur during the transactional execution and will perform a transactional abort if necessary.

Even though the eliding processor did not perform any external write operations to the lock, the hardware ensures program order of operations on the lock. If the eliding processor itself reads the value of the lock in the critical section, it will appear as if the processor had acquired the lock, i.e. the read will return the non-elided value. This behavior makes an HLE execution functionally equivalent to an execution without the HLE prefixes.

The programmer uses the XRELEASE prefix in front of the instruction that is used to release the lock protecting the critical section. This involves a write to the lock. If the instruction is restoring the value of the lock to the value it had prior to the XACQUIRE prefixed lock acquire operation on the same lock, then the processor elides the external write request associated with the release of the lock and does not add the address of the lock to the write-set. The processor then attempts to commit the transactional execution.

With HLE, if multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each other's data, then the threads can execute concurrently and without serialization. Even though the software uses lock acquisition operations on a common lock, the hardware recognizes this, elides the lock, and executes the critical sections on the two threads without requiring any communication through the lock — if such communication was dynamically unnecessary.

If the processor is unable to execute the region transactionally, it will execute the region non-transactionally and without elision. HLE enabled software has the same forward progress guarantees as the underlying non-HLE lock-based execution. For successful HLE execution, the lock and the critical section code must follow certain guidelines (discussed in Section 8.3.3 and Section 8.3.8). These guidelines only affect performance; not following these guidelines will not cause a functional failure.

Hardware without HLE support will ignore the XACQUIRE and XRELEASE prefix hints and will not perform any elision since these prefixes correspond to the REPNE/REPE IA-32 prefixes which are ignored on the instructions where XACQUIRE and XRELEASE are valid. Importantly, HLE is compatible with the existing lock-based programming

model. Improper use of hints will not cause functional bugs though it may expose latent bugs already in the code.

8.2.2 Restricted Transactional Memory

Restricted Transactional Memory (RTM) provides a flexible software interface for transactional execution. RTM provides three new instructions—XBEGIN, XEND, and XABORT—for programmers to start, commit, and abort a transactional execution.

The programmer uses the XBEGIN instruction to specify the start of the transactional code region and the XEND instruction to specify the end of the transactional code region. The XBEGIN instruction takes an operand that provides a relative offset to the **fallback instruction address** if the RTM region could not be successfully executed transactionally.

A processor may abort RTM transactional execution for many reasons. The hardware automatically detects transactional abort conditions and restarts execution from the fallback instruction address with the architectural state corresponding to that at the start of the XBEGIN instruction and the EAX register updated to describe the abort status.

The XABORT instruction allows programmers to abort the execution of an RTM region explicitly. The XABORT instruction takes an 8 bit immediate argument that is loaded into the EAX register and will thus be available to software following an RTM abort.

RTM instructions do not have any data memory location associated with them. While the hardware provides no guarantees as to whether an RTM region will ever successfully commit transactionally, most transactions that follow the recommended guidelines (See Section 8.3.8) are expected to successfully commit transactionally. However, programmers must always provide an alternative code sequence in the fallback path to guarantee forward progress. This may be as simple as acquiring a lock and executing the specified code region non-transactionally. Further, a transaction that always aborts on a given implementation may complete transactionally on a future implementation. Therefore, programmers must ensure the code paths for the transactional region and the alternative code sequence are functionally tested.

8.3 INTEL® TSX APPLICATION PROGRAMMING MODEL

8.3.1 Detection of Transactional Synchronization Support

8.3.1.1 Detection of HLE Support

A processor supports HLE execution if CPUID.07H.EBX.HLE [bit 4] = 1. However, an application can use the HLE prefixes (XACQUIRE and XRELEASE) without checking

whether the processor supports HLE. Processors without HLE support ignore these prefixes and will execute the code without entering transactional execution.

8.3.1.2 Detection of RTM Support

A processor supports RTM execution if CPUID.07H.EBX.RTM [bit 11] = 1. An application must check if the processor supports RTM before it uses the RTM instructions (XBEGIN, XEND, XABORT). These instructions will generate a #UD exception when used on a processor that does not support RTM.

8.3.1.3 Detection of XTEST Instruction

A processor supports the XTEST instruction if it supports either HLE or RTM. An application must check either of these feature flags before using the XTEST instruction. This instruction will generate a #UD exception when used on a processor that does not support either HLE or RTM.

8.3.2 Querying Transactional Execution Status

The XTEST instruction can be used to determine the transactional status of a transactional region specified by HLE or RTM. Note, while the HLE prefixes are ignored on processors that do not support HLE, the XTEST instruction will generate a #UD exception when used on processors that do not support either HLE or RTM.

8.3.3 Requirements for HLE Locks

For HLE execution to successfully commit transactionally, the lock must satisfy certain properties and access to the lock must follow certain guidelines.

- An XRELEASE prefixed instruction must restore the value of the elided lock to the value it had before the lock acquisition. This allows hardware to safely elide locks by not adding them to the write-set. The data size and data address of the lock release (XRELEASE prefixed) instruction must match that of the lock acquire (XACQUIRE prefixed) and the lock must not cross a cache line boundary.
- Software should not write to the elided lock inside a transactional HLE region with any instruction other than an XRELEASE prefixed instruction, otherwise it may cause a transactional abort. In addition, recursive locks (where a thread acquires the same lock multiple times without first releasing the lock) may also cause a transactional abort. Note that software can observe the result of the elided lock acquire inside the critical section. Such a read operation will return the value of the write to the lock.

The processor automatically detects violations to these guidelines, and safely transitions to a non-transactional execution without elision. Since Intel TSX detects conflicts at the granularity of a cache line, writes to data collocated on the same

cache line as the elided lock may be detected as data conflicts by other logical processors eliding the same lock.

8.3.4 Transactional Nesting

Both HLE and RTM support nested transactional regions. However, a transactional abort restores state to the operation that started transactional execution: either the outermost XACQUIRE prefixed HLE eligible instruction or the outermost XBEGIN instruction. The processor treats all nested transactions as one monolithic transaction.

8.3.4.1 HLE Nesting and Elision

Programmers can nest HLE regions up to an implementation specific depth of MAX_HLE_NEST_COUNT. Each logical processor tracks the nesting count internally but this count is not available to software. An XACQUIRE prefixed HLE-eligible instruction increments the nesting count, and an XRELEASE prefixed HLE-eligible instruction decrements it. The logical processor enters transactional execution when the nesting count goes from zero to one. The logical processor attempts to commit only when the nesting count becomes zero. A transactional abort may occur if the nesting count exceeds MAX_HLE_NEST_COUNT.

In addition to supporting nested HLE regions, the processor can also elide multiple nested locks. The processor tracks a lock for elision beginning with the XACQUIRE prefixed HLE eligible instruction for that lock and ending with the XRELEASE prefixed HLE eligible instruction for that same lock. The processor can, at any one time, track up to a MAX_HLE_ELIDED_LOCKS number of locks. For example, if the implementation supports a MAX_HLE_ELIDED_LOCKS value of two and if the programmer nests three HLE identified critical sections (by performing XACQUIRE prefixed HLE eligible instructions on three distinct locks without performing an intervening XRELEASE prefixed HLE eligible instruction on any one of the locks), then the first two locks will be elided, but the third won't be elided (but will be added to the transaction’s write-set). However, the execution will still continue transactionally. Once an XRELEASE for one of the two elided locks is encountered, a subsequent lock acquired through the XACQUIRE prefixed HLE eligible instruction will be elided.

The processor attempts to commit the HLE execution when all elided XACQUIRE and XRELEASE pairs have been matched, the nesting count goes to zero, and the locks have satisfied the requirements described earlier. If execution cannot commit atomically, then execution transitions to a non-transactional execution without elision as if the first instruction did not have an XACQUIRE prefix.

8.3.4.2 RTM Nesting

Programmers can nest RTM regions up to an implementation specific MAX_RTM_NEST_COUNT. The logical processor tracks the nesting count internally but this count is not available to software. An XBEGIN instruction increments the

nesting count, and an XEND instruction decrements it. The logical processor attempts to commit only if the nesting count becomes zero. A transactional abort occurs if the nesting count exceeds MAX_RTM_NEST_COUNT.

8.3.4.3 Nesting HLE and RTM

HLE and RTM provide two alternative software interfaces to a common transactional execution capability. The behavior when HLE and RTM are nested together—HLE inside RTM or RTM inside HLE—is implementation specific. However, in all cases, the implementation will maintain HLE and RTM semantics. An implementation may choose to ignore HLE hints when used inside RTM regions, and may cause a transactional abort when RTM instructions are used inside HLE regions. In the latter case, the transition from transactional to non-transactional execution occurs seamlessly since the processor will re-execute the HLE region without actually doing elision, and then execute the RTM instructions.

8.3.5 RTM Abort Status Definition

RTM uses the EAX register to communicate abort status to software. Following an RTM abort the EAX register has the following definition.

Table 8-1. RTM Abort Status Definition

EAX Register Bit Position	Meaning
0	Set if abort caused by XABORT instruction.
1	If set, the transaction may succeed on a retry. This bit is always clear if bit 0 is set.
2	Set if another logical processor conflicted with a memory address that was part of the transaction that aborted.
3	Set if an internal buffer overflowed.
4	Set if a debug breakpoint was hit.
5	Set if an abort occurred during execution of a nested transaction.
23:6	Reserved.
31:24	XABORT argument (only valid if bit 0 set, otherwise reserved).

The EAX abort status for RTM only provides causes for aborts. It does not by itself encode whether an abort or commit occurred for the RTM region. The value of EAX can be 0 following an RTM abort. For example, a CPUID instruction when used inside an RTM region causes a transactional abort and may not satisfy the requirements for setting any of the EAX bits. This may result in an EAX value of 0.

8.3.6 RTM Memory Ordering

A successful RTM commit causes all memory operations in the RTM region to appear to execute atomically. A successfully committed RTM region consisting of an XBEGIN followed by an XEND, even with no memory operations in the RTM region, has the same ordering semantics as a LOCK prefixed instruction.

The XBEGIN instruction does not have fencing semantics. However, if an RTM execution aborts, all memory updates from within the RTM region are discarded and never made visible to any other logical processor.

8.3.7 RTM-Enabled Debugger Support

By default, any debug exception inside an RTM region will cause a transactional abort and will redirect control flow to the fallback instruction address with architectural state recovered and bit 4 in EAX set. However, to allow software debuggers to intercept execution on debug exceptions, the RTM architecture provides additional capability.

If bit 11 of DR7 and bit 15 of the IA32_DEBUGCTL_MSR are both 1, any RTM abort due to a debug exception (#DB) or breakpoint exception (#BP) causes execution to roll back and restart from the XBEGIN instruction instead of the fallback address. In this scenario, the EAX register will also be restored back to the point of the XBEGIN instruction.

8.3.8 Programming Considerations

Typical programmer-identified regions are expected to transactionally execute and commit successfully. However, Intel TSX does not provide any such guarantee. A transactional execution may abort for many reasons. To take full advantage of the transactional capabilities, programmers should follow certain guidelines to increase the probability of their transactional execution committing successfully.

This section discusses various events that may cause transactional aborts. The architecture ensures that updates performed within a transaction that subsequently aborts execution will never become visible. Only a committed transactional execution updates architectural state. Transactional aborts never cause functional failures and only affect performance.

8.3.8.1 Instruction Based Considerations

Programmers can use any instruction safely inside a transaction (HLE or RTM) and can use transactions at any privilege level. However, some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path.

Intel TSX allows for most common instructions to be used inside transactions without causing aborts. The following operations inside a transaction do not typically cause an abort.

- Operations on the instruction pointer register, general purpose registers (GPRs) and the status flags (CF, OF, SF, PF, AF, and ZF).
- Operations on XMM and YMM registers and the MXCSR register

However, programmers must be careful when intermixing SSE and AVX operations inside a transactional region. Intermixing SSE instructions accessing XMM registers and AVX instructions accessing YMM registers may cause transactions to abort.

Programmers may use REP/REPNE prefixed string operations inside transactions. However, long strings may cause aborts. Further, the use of CLD and STD instructions may cause aborts if they change the value of the DF flag. However, if DF is 1, the STD instruction will not cause an abort. Similarly, if DF is 0, the CLD instruction will not cause an abort.

Instructions not enumerated here as causing abort when used inside a transaction will typically not cause a transaction to abort (examples include but are not limited to MFENCE, LFENCE, SFENCE, RDTSC, RDTSCP, etc.).

The following instructions will abort transactional execution on any implementation:

- XABORT
- CPUID
- PAUSE

In addition, in some implementations, the following instructions may always cause transactional aborts. These instructions are not expected to be commonly used inside typical transactional regions. However, programmers must not rely on these instructions to force a transactional abort, since whether they cause transactional aborts is implementation dependent.

- Operations on X87 and MMX architecture state. This includes all MMX and X87 instructions, including the FXRSTOR and FXSAVE instructions.
- Update to non-status portion of EFLAGS: CLI, STI, POPFD, POPFQ, CLTS.
- Instructions that update segment registers, debug registers and/or control registers: MOV to DS/ES/FS/GS/SS, POP DS/ES/FS/GS/SS, LDS, LES, LFS, LGS, LSS, SWAPGS, WRFSBASE, WRGSBASE, LGDT, SGDT, LIDT, SIDT, LLDT, SLDT, LTR, STR, Far CALL, Far JMP, Far RET, IRET, MOV to DRx, MOV to CR0/CR2/CR3/CR4/CR8 and LMSW.
- Ring transitions: SYSENTER, SYSCALL, SYSEXIT, and SYSRET.
- TLB and Cacheability control: CLFLUSH, INVD, WBINVD, INVLPG, INVPCID, and memory instructions with a non-temporal hint (MOVNTDQA, MOVNTDQ, MOVNTI, MOVNTPD, MOVNTPS, and MOVNTQ).
- Processor state save: XSAVE, XSAVEOPT, and XRSTOR.
- Interrupts: INTn, INTO.
- IO: IN, INS, REP INS, OUT, OUTS, REP OUTS and their variants.

- VMX: VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, and INVVPID.
- SMX: GETSEC.
- UD2, RSM, RDMSR, WRMSR, HLT, MONITOR, MWAIT, XSETBV, VZEROUPPER, MASKMOVQ, and V/MASKMOVDQU.

8.3.8.2 Runtime Considerations

In addition to the instruction-based considerations, runtime events may cause transactional execution to abort. These may be due to data access patterns or micro-architectural implementation causes. Keep in mind that the following list is not a comprehensive discussion of all abort causes.

Any fault or trap in a transaction that must be exposed to software will be suppressed. Transactional execution will abort and execution will transition to a non-transactional execution, as if the fault or trap had never occurred. If any exception is not masked, that will result in a transactional abort and it will be as if the exception had never occurred.

Synchronous exception events (#DE, #OF, #NP, #SS, #GP, #BR, #UD, #AC, #XF, #PF, #NM, #TS, #MF, #DB, #BP/INT3) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred. With HLE, since the non-transactional code path is identical to the transactional code path, these events will typically re-appear when the instruction that caused the exception is re-executed non-transactionally, causing the associated synchronous events to be delivered appropriately in the non-transactional execution.

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a non-transactional execution. The asynchronous events will be pended and handled after the transactional abort is processed.

Transactions only support write-back cacheable memory type operations. A transaction may always abort if it includes operations on any other memory type. This includes instruction fetches to UC memory type.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. The behavior of how the processor handles this is implementation specific. Some implementations may allow the updates to these flags to become externally visible even if the transactional region subsequently aborts. Some Intel TSX implementations may choose to abort the transactional execution if these flags need to be updated. Further, a processor's page-table walk may generate accesses to its own transactionally written but uncommitted state. Some Intel TSX implementations may choose to abort the execution of a transactional region in such situations. Regardless, the architecture ensures that, if the transactional region aborts, then the transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs.

Executing self-modifying code transactionally may also cause transactional aborts. Programmers must continue to follow the Intel recommended guidelines for writing self-modifying and cross-modifying code even when employing HLE and RTM.

While an implementation of RTM and HLE will typically provide sufficient resources for executing common transactional regions, implementation constraints and excessive sizes for transactional regions may cause a transactional execution to abort and transition to a non-transactional execution. The architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed.

Conflicting requests to a cache line accessed within a transactional region may prevent the transaction from executing successfully. For example, if logical processor P0 reads line A in a transactional region and another logical processor P1 writes A (either inside or outside a transactional region) then logical processor P0 may abort if logical processor P1’s write interferes with processor P0's ability to execute transactionally. Similarly, if P0 writes line A in a transactional region and P1reads or writes A (either inside or outside a transactional region), then P0 may abort if P1's access to A interferes with P0's ability to execute transactionally. In addition, other coherence traffic may at times appear as conflicting requests and may cause aborts. While these false conflicts may happen, they are expected to be uncommon. The conflict resolution policy to determine whether P0 or P1 aborts in the above scenarios is implementation specific.

8.4 INSTRUCTION REFERENCE

Conventions and notations of instruction format can be found in Section 5.1.

XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE[1]	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an "XRELEASE-enabled" instruction to end lock elision on the instruction memory operand address.

NOTES:

1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in Section 8.3.3 for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE

```
IF XACQUIRE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
            HLE_NEST_COUNT++
            IF (HLE_NEST_COUNT = 1) THEN
                HLE_ACTIVE ← 1
                IF 64-bit mode
                    THEN
                        restartRIP ← instruction pointer of the XACQUIRE-enabled instruction
                    ELSE
                        restartEIP ← instruction pointer of the XACQUIRE-enabled instruction
                FI;
                Enter HLE Execution (* record register state, start tracking memory state *)
            FI; (* HLE_NEST_COUNT = 1*)
            IF ElisionBufferAvailable
                THEN
```

```
                    Allocate elision buffer
                    Record address and data for forwarding and commit checking
                    Perform elision
                ELSE
                    Perform lock acquire operation transactionally but without elision
            FI;
        ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
                GOTO HLE_ABORT_PROCESSING
        FI;
    ELSE
        Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
FI;
```

XRELEASE

```
IF XRELEASE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT > 0)
            THEN
                HLE_NEST_COUNT--
                IF lock address matches in elision buffer THEN
                    IF lock satisfies address and value requirements THEN
                        Deallocate elision buffer
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI;
                IF (HLE_NEST_COUNT = 0)
                    THEN
                        IF NoAllocatedElisionBuffer
                            THEN
                                Try to commit transaction
                                IF fail to commit transaction
                                    THEN
                                        GOTO HLE_ABORT_PROCESSING;
                                    ELSE (* commit success *)
                                        HLE_ACTIVE ← 0
                                FI;
                            ELSE
                                GOTO HLE_ABORT_PROCESSING
                        FI;
                FI;
        FI; (* HLE_NEST_COUNT > 0 *)
```

```
    ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE ← 0
    HLE_NEST_COUNT ← 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP ← restartRIP
        ELSE
            EIP ← restartEIP
    FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
```

SIMD Floating-Point Exceptions

None

Other Exceptions

#GP(0)	If the use of prefix causes instruction length to exceed 15 bytes.

XABORT — Transaction Abort

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

Operation

XABORT

```
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XABORT: void _xabort(unsigned int);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK prefix is used.

XBEGIN — Transaction Begin

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

A relative offset (rel16 or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP (or RIP) register. (Here, the EIP (or RIP) register contains the address of the instruction following the XBEGIN instruction).

Operation

XBEGIN

```
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
    THEN
```

```
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                        tempRIP ← RIP + SignExtend(IMM)
                                (* RIP is instruction following XBEGIN instruction *)
                    ELSE
                        tempEIP ← EIP + SignExtend(IMM)
                                (* EIP is instruction following XBEGIN instruction *)
            FI;

            IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
            and tempEIP outside code segment limit
                THEN #GP(0);
            FI;
            IF 64-bit mode and tempRIP is not canonical
                THEN #GP(0);
            FI;
            IF 64-bit Mode
                THEN
                    fallbackRIP ← tempRIP
                ELSE
                    IF 16-bit Mode
                        THEN
                            fallbackEIP ← tempEIP AND 0000FFFFH
                        ELSE (* 32-bit mode *)
                            fallbackEIP ← tempEIP
                    FI;
            FI;
            RTM_ACTIVE ← 1
            Enter RTM Execution (* record register state, start tracking memory state*)
        FI; (* RTM_NEST_COUNT = 1 *)
    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
        GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
```

```
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN: unsigned int _xbegin(void);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11]=0. If LOCK prefix is used.
#GP(0)	If the fallback address is outside the CS segment.

Real-Address Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11]=0. If LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11]=0. If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK prefix is used.

#GP(0)	If the fallback address is non-canonical.

XEND — Transaction End

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
0F 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transaction will cause a #GP (General Protection Fault).

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transaction
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE ← 0
        FI;
    FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
```

```
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XEND: void _xend(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK or 66H or F2H or F3H prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
0F 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST: int _xtest(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK or 66H or F2H or F3H prefix is used.

CHAPTER 9
ADDITIONAL NEW INSTRUCTIONS

This chapter describes additional new instructions for future Intel 64 processors that provide enhancements in selected application domains: ranging from random number generation to multi-precision arithmetic.

9.1 DETECTION OF NEW INSTRUCTIONS

Hardware support for flag-preserving add-carry instructions is indicated by the following feature flags:

- CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19]=1 indicates the processor supports ADCX and ADOX instructions.
- Hardware support for the RDSEED instruction is indicated by CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18].
- CPUID.(EAX=8000_0001H):ECX[bit 8]=1 indicates PREFETCHW is supported.

9.2 RANDOM NUMBER INSTRUCTIONS

The instructions for generating random numbers to comply with NIST SP800-90A, SP800-90B, and SP800-90C standards are described in this section.

9.2.1 RDRAND

The RDRAND instruction returns a random number. All Intel processors that support the RDRAND instruction indicate the availability of the RDRAND instruction via reporting CPUID.01H:ECX.RDRAND[bit 30] = 1.

RDRAND returns random numbers that are supplied by a cryptographically secure, deterministic random bit generator (DRBG). The DRBG is designed to meet the NIST SP 800-90A standard. The DRBG is re-seeded frequently from a on-chip non-deterministic entropy source to guarantee data returned by RDRAND is statistically uniform, non-periodic and non-deterministic.

In order for the hardware design to meet its security goals, the random number generator continuously tests itself and the random data it is generating. Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDRAND instruction will return no data instead of bad data.

Under heavy load, with multiple cores executing RDRAND in parallel, it is possible, though unlikely, for the demand of random numbers by software processes/threads

to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDRAND instruction returning no data transitorily. The RDRAND instruction indicates the occurrence of this rare situation by clearing the CF flag.

The RDRAND instruction returns with the carry flag set (CF = 1) to indicate valid data is returned. It is recommended that software using the RDRAND instruction to get random numbers retry for a limited number of iterations while RDRAND returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows. A retry limit should be employed to prevent a hard failure in the RNG (expected to be extremely rare) leading to a busy loop in software.

The intrinsic primitive for RDRAND is defined to address software's need for the common cases (CF = 1) and the rare situations (CF = 0). The intrinsic primitive returns a value that reflects the value of the carry flag returned by the underlying RDRAND instruction. The example below illustrates the recommended usage of an RDRAND instrinsic in a utility function, a loop to fetch a 64-bit random value with a retry count limit of 10. A C implementation might be written as follows:

```
-------------------------------------------------------------------------------------------
#define SUCCESS 1
#define RETRY_LIMIT_EXCEEDED 0
#define RETRY_LIMIT 10


int get_random_64( unsigned __int 64 * arand)
{int i     ;
    for ( i = 0; i < RETRY_LIMIT; i ++) {
          if(_rdrand64_step(arand) ) return SUCCESS;
    }
    return RETRY_LIMIT_EXCEEDED;
}
-------------------------------------------------------------------------------
```

The RDRAND instruction is first introduced in third-generation Intel Core processors built on 22-nm process.

9.2.2 RDSEED

The RDSEED instruction returns a random number. All Intel processors that support the RDSEED instruction indicate the availability of the RDSEED instruction via reporting CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 1.

RDSEED returns random numbers that are supplied by a cryptographically secure, enhanced non-deterministic random bit generator (Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP800-90C standards.

In order for the hardware design to meet its security goals, the random number generator continuously tests itself and the random data it is generating. Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDSEED instruction will return no data instead of bad data.

Under heavy load, with multiple cores executing RDSEED in parallel, it is possible for the demand of random numbers by software processes/threads to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDSEED instruction returning no data transitorily. The RDSEED instruction indicates the occurrence of this situation by clearing the CF flag.

The RDSEED instruction returns with the carry flag set (CF = 1) to indicate valid data is returned. It is recommended that software using the RDSEED instruction to get random numbers retry for a limited number of iterations while RDSEED returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows. A retry limit should be employed to prevent a hard failure in the NRBG (expected to be extremely rare) leading to a busy loop in software.

The intrinsic primitive for RDSEED is defined to address software's need for the common cases (CF = 1) and the rare situations (CF = 0). The intrinsic primitive returns a value that reflects the value of the carry flag returned by the underlying RDRAND instruction. The example below illustrates the recommended usage of an RDRAND instrinsic in a utility function, a loop to fetch a 64 bit random value with a retry count limit of 10.

9.2.3 RDSEED and VMX interactions

A VM-execution control exists that allows the virtual machine monitor to trap on the instruction. The "RDSEED exiting" VM-execution control is located at bit 16 of the secondary processor-based VM-execution controls. A VM exit due to RDSEED will have exit reason 61 (decimal).

9.3 INSTRUCTION EXCEPTION SPECIFICATION

To use this reference of instruction exceptions, look at each instruction for a description of the particular exception type of interest. 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 9-1 lists exception conditions for ADCX and ADOX.

Table 9-1. Exception Definition (ADCX and ADOX Instructions)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If ADX CPUID feature flag is '0'
	X	X	X	X	If a LOCK prefix is present
Stack, SS(0)	X	X	X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

9.4 INSTRUCTION FORMAT

The format used for describing each instruction as in the example below is described in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag (THIS IS AN EXAMPLE)

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

9.5 INSTRUCTION SET REFERENCE

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

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

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

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

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

ADCX executes normally either inside or outside a transaction region.

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

Operation

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

Flags Affected

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

Intel C/C++ Compiler Intrinsic Equivalent

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 9-1

ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

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

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

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

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

ADOX executes normally either inside or outside a transaction region.

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

Operation

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

Flags Affected

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

Intel C/C++ Compiler Intrinsic Equivalent

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 9-1

PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /1 PREFETCHW m8	A	V/V	PRFCHW	Move data from m8 closer to the processor in anticipation of a write.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates all other cached instances of the line.

The source operand is a byte memory location. (Use of any ModR/M value other than a memory operand will lead to unpredictable behavior.) If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates any other cached copy in anticipation of the line being written to in the future.

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

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH instructions, other PREFETCHW instructions, or any other general instruction

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

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

Operation

FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

void _m_prefetchw(void *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /7 RDSEED r16	A	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
0F C7 /7 RDSEED r32	A	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
REX.W + 0F C7 /7 RDSEED r64	A	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:r/m (w)	NA	NA	NA

Description

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

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

Operation

```
IF HW_NRND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_NRND_GEN.data;
            osize is 32: DEST[31:0] ← HW_NRND_GEN.data;
            osize is 16: DEST[15:0] ← HW_NRND_GEN.data;
        ESAC;
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC;
        CF ← 0;
FI;

OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

RDSEED int _rdseed16_step(unsigned short *);

RDSEED int _rdseed32_step(unsigned int *);

RDSEED int _rdseed64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0

APPENDIX A
INSTRUCTION SUMMARY

A.1 AVX INSTRUCTIONS

In AVX, 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 in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
yes	yes	YY 0F 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 VPERMILPS
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 VPERMILPS
yes	yes		MOVAPS	
yes	yes		MOVSHDUP	
yes	yes		MOVAPD	

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
no	no		CVTPI2PS	MMX
no	yes		CVTSI2SS	scalar
no	no		CVTPI2PD	MMX
no	yes		CVTSI2SD	scalar
yes	yes		MOVNTPS	
yes	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
no	yes		UCOMISS	scalar
no	yes		UCOMISD	scalar
no	yes		COMISS	scalar
no	yes		COMISD	scalar
yes	yes	YY 0F 5X	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	

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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		CVTPS2PD	
no	yes		CVTSS2SD	scalar
yes	yes		CVTPD2PS	
no	yes		CVTSD2SS	scalar
yes	yes		CVTDQ2PS	
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		MINSD	scalar
yes	yes		DIVPS	

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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 0F 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		MOVD	scalar
no	yes		MOVQ	scalar
yes	yes		MOVDQA	
yes	yes		MOVDQU	
no	yes	YY 0F 7X	PSHUFD	VI
no	yes		PSHUFHW	VI
no	yes		PSHUFLW	VI
no	yes		PCMPEQB	VI
no	yes		PCMPEQW	VI

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
no	yes		PCMPEQD	VI
yes	yes		HADDPD	
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 0F AX	LDMXCSR	
no	yes		STMXCSR	
yes	yes	YY 0F 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 0F 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		PMOVMSKB	VI
no	yes		PSUBUSB	VI

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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 0F 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	
yes	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 0F FX	LDDQU	VI
no	yes		PSLLW	VI
no	yes		PSLLD	VI
no	yes		PSLLQ	VI
no	yes		PMULUDQ	VI

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
no	yes		PMADDWD	VI
no	yes		PSADBW	VI
no	yes		MASKMOVDQU	
no	yes		PSUBB	VI
no	yes		PSUBW	VI
no	yes		PSUBD	VI
no	yes		PSUBQ	VI
no	yes		PADDB	VI
no	yes		PADDW	VI
no	yes		PADDD	VI
no	yes	SSSE3	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

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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
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

Table A-1. Promoted SSE/SSE2/SSE3/SSSE3/SSE4 Instructions in AVX

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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	POPCNT	POPCNT	integer
no	yes	AESNI	AESDEC	VI
no	yes		AESDECLAST	VI
no	yes		AESENC	VI
no	yes		AESECNLAST	VI
no	yes		AESIMC	VI
no	yes		AESKEYGENASSIST	VI
no	yes	CLMUL	PCLMULQDQ	VI

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: MOVLPD/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: BLENDVPD and BLENDVPS are superseded by the more flexible VBLENDVPD 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).

A.2 PROMOTED VECTOR INTEGER INSTRUCTIONS IN AVX2

In AVX2, most SSE/SSE2/SSE3/SSSE3/SSE4 vector integer instructions have been promoted to support VEX.256 encodings. Table A-2 summarizes the promotion status for existing instructions. The column "VEX.128" indicates whether the instruction using VEX.128 prefix encoding is supported.

The column "VEX.256" indicates whether 256-bit vector form of the instruction using the VEX.256 prefix encoding is supported, and under which feature flag.

Table A-2. Promoted Vector Integer SIMD Instructions in AVX2

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction
AVX2	AVX	YY 0F 6X	PUNPCKLBW
AVX2	AVX		PUNPCKLWD
AVX2	AVX		PUNPCKLDQ
AVX2	AVX		PACKSSWB
AVX2	AVX		PCMPGTB
AVX2	AVX		PCMPGTW
AVX2	AVX		PCMPGTD
AVX2	AVX		PACKUSWB
AVX2	AVX		PUNPCKHBW
AVX2	AVX		PUNPCKHWD
AVX2	AVX		PUNPCKHDQ
AVX2	AVX		PACKSSDW
AVX2	AVX		PUNPCKLQDQ
AVX2	AVX		PUNPCKHQDQ
no	AVX		MOVD
no	AVX		MOVQ
AVX	AVX		MOVDQA
AVX	AVX		MOVDQU
AVX2	AVX	YY 0F 7X	PSHUFD
AVX2	AVX		PSHUFHW
AVX2	AVX		PSHUFLW
AVX2	AVX		PCMPEQB

Table A-2. Promoted Vector Integer SIMD Instructions in AVX2

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction
AVX2	AVX		PCMPEQW
AVX2	AVX		PCMPEQD
AVX	AVX		MOVDQA
AVX	AVX		MOVDQU
no	AVX		PINSRW
no	AVX		PEXTRW
AVX2	AVX		PSRLW
AVX2	AVX		PSRLD
AVX2	AVX		PSRLQ
AVX2	AVX		PADDQ
AVX2	AVX		PMULLW
AVX2	AVX		PMOVMSKB
AVX2	AVX		PSUBUSB
AVX2	AVX		PSUBUSW
AVX2	AVX		PMINUB
AVX2	AVX		PAND
AVX2	AVX		PADDUSB
AVX2	AVX		PADDUSW
AVX2	AVX		PMAXUB
AVX2	AVX		PANDN
AVX2	AVX	YY 0F EX	PAVGB
AVX2	AVX		PSRAW
AVX2	AVX		PSRAD
AVX2	AVX		PAVGW
AVX2	AVX		PMULHUW
AVX2	AVX		PMULHW
AVX	AVX		MOVNTDQ
AVX2	AVX		PSUBSB
AVX2	AVX		PSUBSW
AVX2	AVX		PMINSW

Table A-2. Promoted Vector Integer SIMD Instructions in AVX2

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction
AVX2	AVX		POR
AVX2	AVX		PADDSB
AVX2	AVX		PADDSW
AVX2	AVX		PMAXSW
AVX2	AVX		PXOR
AVX	AVX	YY 0F FX	LDDQU
AVX2	AVX		PSLLW
AVX2	AVX		PSLLD
AVX2	AVX		PSLLQ
AVX2	AVX		PMULUDQ
AVX2	AVX		PMADDWD
AVX2	AVX		PSADBW
AVX2	AVX		PSUBB
AVX2	AVX		PSUBW
AVX2	AVX		PSUBD
AVX2	AVX		PSUBQ
AVX2	AVX		PADDB
AVX2	AVX		PADDW
AVX2	AVX		PADDD
AVX2	AVX	SSSE3	PHADDW
AVX2	AVX		PHADDSW
AVX2	AVX		PHADDD
AVX2	AVX		PHSUBW
AVX2	AVX		PHSUBSW
AVX2	AVX		PHSUBD
AVX2	AVX		PMADDUBSW
AVX2	AVX		PALIGNR
AVX2	AVX		PSHUFB
AVX2	AVX		PMULHRSW
AVX2	AVX		PSIGNB

Table A-2. Promoted Vector Integer SIMD Instructions in AVX2

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction
AVX2	AVX		PSIGNW
AVX2	AVX		PSIGND
AVX2	AVX		PABSB
AVX2	AVX		PABSW
AVX2	AVX		PABSD
AVX2	AVX		MOVNTDQA
AVX2	AVX		MPSADBW
AVX2	AVX		PACKUSDW
AVX2	AVX		PBLENDVB
AVX2	AVX		PBLENDW
AVX2	AVX		PCMPEQQ
no	AVX		PEXTRD
no	AVX		PEXTRQ
no	AVX		PEXTRB
no	AVX		PEXTRW
no	AVX		PHMINPOSUW
no	AVX		PINSRB
no	AVX		PINSRD
no	AVX		PINSRQ
AVX2	AVX		PMAXSB
AVX2	AVX		PMAXSD
AVX2	AVX		PMAXUD
AVX2	AVX		PMAXUW
AVX2	AVX		PMINSB
AVX2	AVX		PMINSD
AVX2	AVX		PMINUD
AVX2	AVX		PMINUW
AVX2	AVX		PMOVSXxx
AVX2	AVX		PMOVZXxx
AVX2	AVX		PMULDQ

Table A-2. Promoted Vector Integer SIMD Instructions in AVX2

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction
AVX2	AVX		PMULLD
AVX	AVX		PTEST
AVX2	AVX	SSE4.2	PCMPGTQ
no	AVX		PCMPESTRI
no	AVX		PCMPESTRM
no	AVX		PCMPISTRI
no	AVX		PCMPISTRM
no	AVX	AESNI	AESDEC
no	AVX		AESDECLAST
no	AVX		AESENC
no	AVX		AESECNLAST
no	AVX		AESIMC
no	AVX		AESKEYGENASSIST
no	AVX	CLMUL	PCLMULQDQ

Table A-3 compares complementary SIMD functionalities introduced in AVX and AVX2. instructions.

Table A-3. VEX-Only SIMD Instructions in AVX and AVX2

AVX2	AVX	Comment
VBROADCASTI128	VBROADCASTF128	256-bit only
VBROADCASTSD ymm1, xmm	VBROADCASTSD ymm1, m64	256-bit only
VBROADCASTSS (from xmm)	VBROADCASTSS (from m32)	
VEXTRACTI128	VEXTRACTF128	256-bit only
VINSERTI128	VINSERTF128	256-bit only
VPMASKMOVD	VMASKMOVPS	
VPMASKMOVQ!	VMASKMOVPD	
	VPERMILPD	in-lane
	VPERMILPS	in-lane
VPERM2I128	VPERM2F128	256-bit only
VPERMD		cross-lane

Table A-3. VEX-Only SIMD Instructions in AVX and AVX2

AVX2	AVX	Comment
VPERMPS		cross-lane
VPERMQ		cross-lane
VPERMPD		cross-lane
	VTESTPD	
	VTESTPS	
VPBLENDD		
VPSLLVD/Q		
VPSRAVD		
VPSRLVD/Q		
VGATHERDPD/QPD		
VGATHERDPS/QPS		
VPGATHERDD/QD		
VPGATHERDQ/QQ		

Table A-4. New Primitive in AVX2 Instructions

Opcode	Instruction	Description
VEX.NDS.256.66.0 F38.W0 36 /r	VPERMD ymm1, ymm2, ymm3/m256	Permute doublewords in ymm3/m256 using indexes in ymm2 and store the result in ymm1.
VEX.NDS.256.66.0 F3A.W1 01 /r	VPERMPD ymm1, ymm2/m256, imm8	Permute double-precision FP elements in ymm2/m256 using indexes in imm8 and store the result in ymm1.
VEX.NDS.256.66.0 F38.W0 16 /r	VPERMPS ymm1, ymm2, ymm3/m256	Permute single-precision FP elements in ymm3/m256 using indexes in ymm2 and store the result in ymm1.
VEX.NDS.256.66.0 F3A.W1 00 /r	VPERMQ ymm1, ymm2/m256, imm8	Permute quadwords in ymm2/m256 using indexes in imm8 and store the result in ymm1.
VEX.NDS.128.66.0 F38.W0 47 /r	VPSLLVD xmm1, xmm2, xmm3/m128	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0 F38.W1 47 /r	VPSLLVQ xmm1, xmm2, xmm3/m128	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.

Opcode	Instruction	Description
VEX.NDS.256.66.0 F38.W0 47 /r	VPSLLVD ymm1, ymm2, ymm3/m256	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0 F38.W1 47 /r	VPSLLVQ ymm1, ymm2, ymm3/m256	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.128.66.0 F38.W0 46 /r	VPSRAVD xmm1, xmm2, xmm3/m128	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in the sign bits.
VEX.NDS.128.66.0 F38.W0 45 /r	VPSRLVD xmm1, xmm2, xmm3/m128	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0 F38.W1 45 r	VPSRLVQ xmm1, xmm2, xmm3/m128	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0 F38.W0 45 /r	VPSRLVD ymm1, ymm2, ymm3/m256	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0 F38.W1 45 /r	VPSRLVQ ymm1, ymm2, ymm3/m256	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.DDS.128.66.0 F38.W0 90 /r	VGATHERDD xmm1, vm32x, xmm2	Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0 F38.W0 91 /r	VGATHERQD xmm1, vm64x, xmm2	Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0 F38.W0 90 /r	VGATHERDD ymm1, vm32y, ymm2	Using dword indices specified in vm32y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

Opcode	Instruction	Description
VEX.DDS.256.66.0 F38.W0 91 /r	VGATHERQD ymm1, vm64y, ymm2	Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.128.66.0 F38.W1 92 /r	VGATHERDPD xmm1, vm32x, xmm2	Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0 F38.W1 93 /r	VGATHERQPD xmm1, vm64x, xmm2	Using qword indices specified in vm64x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0 F38.W1 92 /r	VGATHERDPD ymm1, vm32x, ymm2	Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.256.66.0 F38.W1 93 /r	VGATHERQPD ymm1, vm64y ymm2	Using qword indices specified in vm64y, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.128.66.0 F38.W0 92 /r	VGATHERDPS xmm1, vm32x, xmm2	Using dword indices specified in vm32x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0 F38.W0 93 /r	VGATHERQPS xmm1, vm64x, xmm2	Using qword indices specified in vm64x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0 F38.W0 92 /r	VGATHERDPS ymm1, vm32y, ymm2	Using dword indices specified in vm32y, gather single-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

Opcode	Instruction	Description
VEX.DDS.256.66.0 F38.W0 93 /r	VGATHERQPS ymm1, vm64y, ymm2	Using qword indices specified in vm64y, gather single-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.128.66.0 F38.W1 90 /r	VGATHERDQ xmm1, vm32x, xmm2	Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.128.66.0 F38.W1 91 /r	VGATHERQQ xmm1, vm64x, xmm2	Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.
VEX.DDS.256.66.0 F38.W1 90 /r	VGATHERDQ ymm1, vm32x, ymm2	Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.
VEX.DDS.256.66.0 F38.W1 91 /r	VGATHERQQ ymm1, vm64y, ymm2	Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.

Table A-5. FMA Instructions

Opcode	Instruction	Description
VEX.DDS.128.66.0 F38.W1 98 /r	VFMADD132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 A8 /r	VFMADD213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 B8 /r	VFMADD231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W1 98 /r	VFMADD132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0 F38.W1 A8 /r	VFMADD213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 B8 /r	VFMADD231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W0 98 /r	VFMADD132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 A8 /r	VFMADD213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 B8 /r	VFMADD231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W0 98 /r	VFMADD132PS ymm0, ymm1, ymm2/m256, ymm3	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add to ymm1 and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 A8 /r	VFMADD213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, add to ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 B8 /r	VFMADD231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W1 99 /r	VFMADD132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 A9 /r	VFMADD213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 B9 /r	VFMADD231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 99 /r	VFMADD132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, add to xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 A9 /r	VFMADD213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, add to xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 B9 /r	VFMADD231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.

Opcode	Instruction	Description
VEX.DDS.128.66.0 F38.W1 96 /r	VFMADDSUB132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, add/subtract elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 A6 /r	VFMADDSUB213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, add/subtract elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 B6 /r	VFMADDSUB231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add/subtract elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W1 96 /r	VFMADDSUB132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, add/subtract elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 A6 /r	VFMADDSUB213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 B6 /r	VFMADDSUB231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm1, add/subtract elements in ymm2/mem and put result in ymm0.
VEX.DDS.128.66.0 F38.W0 96 /r	VFMADDSUB132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, add/subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 A6 /r	VFMADDSUB213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, add/subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 B6 /r	VFMADDSUB231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, add/subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W0 96 /r	VFMADDSUB132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, add/subtract ymm1 and put result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0 F38.W0 A6 /r	VFMADDSUB213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, add/subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 B6 /r	VFMADDSUB231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, add/subtract ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W1 97 /r	VFMSUBADD132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract/add elements in xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 A7 /r	VFMSUBADD213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, subtract/add elements in xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 B7 /r	VFMSUBADD231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract/add elements in xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W1 97 /r	VFMSUBADD132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract/add elements in ymm1 and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 A7 /r	VFMSUBADD213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, subtract/add elements in ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 B7 /r	VFMSUBADD231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract/add elements in ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W0 97 /r	VFMSUBADD132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract/add xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 A7 /r	VFMSUBADD213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, subtract/add xmm2/mem and put result in xmm0.

Opcode	Instruction	Description
VEX.DDS.128.66.0 F38.W0 B7 /r	VFMSUBADD231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract/add xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W0 97 /r	VFMSUBADD132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract/add ymm1 and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 A7 /r	VFMSUBADD213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, subtract/add ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 B7 /r	VFMSUBADD231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract/add ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W1 9A /r	VFMSUB132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 AA /r	VFMSUB213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 BA /r	VFMSUB231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W1 9A /r	VFMSUB132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 AA /r	VFMSUB213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W1 BA /r	VFMSUB231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W0 9A /r	VFMSUB132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 AA /r	VFMSUB213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.

Opcode	Instruction	Description
VEX.DDS.128.66.0 F38.W0 BA /r	VFMSUB231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.256.66.0 F38.W0 9A /r	VFMSUB132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, subtract ymm1 and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 AA /r	VFMSUB213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, subtract ymm2/mem and put result in ymm0.
VEX.DDS.256.66.0 F38.W0 BA /r	VFMSUB231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, subtract ymm0 and put result in ymm0.
VEX.DDS.128.66.0 F38.W1 9B /r	VFMSUB132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 AB /r	VFMSUB213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 BB /r	VFMSUB231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 9B /r	VFMSUB132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, subtract xmm1 and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 AB /r	VFMSUB213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, subtract xmm2/mem and put result in xmm0.
VEX.DDS.128.66.0 F38.W0 BB /r	VFMSUB231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, subtract xmm0 and put result in xmm0.
VEX.DDS.128.66.0 F38.W1 9C /r	VFNMADD132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 AC /r	VFNMADD213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 BC /r	VFNMADD231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0 F38.W1 9C /r	VFNMADD132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W1 AC /r	VFNMADD213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, negate the multiplication result and add to ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W1 BC /r	VFNMADD231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0. Put the result in ymm0.
VEX.DDS.128.66.0 F38.W0 9C /r	VFNMADD132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 AC /r	VFNMADD213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 BC /r	VFNMADD231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.256.66.0 F38.W0 9C /r	VFNMADD132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and add to ymm1. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W0 AC /r	VFNMADD213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, negate the multiplication result and add to ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W0 BC /r	VFNMADD231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and add to ymm0. Put the result in ymm0.
VEX.DDS.128.66.0 F38.W1 9D /r	VFNMADD132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.

Opcode	Instruction	Description
VEX.DDS.128.66.0 F38.W1 AD /r	VFNMADD213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 BD /r	VFNMADD231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 9D /r	VFNMADD132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and add to xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 AD /r	VFNMADD213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, negate the multiplication result and add to xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 BD /r	VFNMADD231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and add to xmm0. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 9E /r	VFNMSUB132PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 AE /r	VFNMSUB213PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 BE /r	VFNMSUB231PD xmm0, xmm1, xmm2/m128	Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.
VEX.DDS.256.66.0 F38.W1 9E /r	VFNMSUB132PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W1 AE /r	VFNMSUB213PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm0, negate the multiplication result and subtract ymm2/mem. Put the result in ymm0.

Opcode	Instruction	Description
VEX.DDS.256.66.0 F38.W1 BE /r	VFNMSUB231PD ymm0, ymm1, ymm2/m256	Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0. Put the result in ymm0.
VEX.DDS.128.66.0 F38.W0 9E /r	VFNMSUB132PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 AE /r	VFNMSUB213PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 BE /r	VFNMSUB231PS xmm0, xmm1, xmm2/m128	Multiply packed single-precision floating-point values from xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.
VEX.DDS.256.66.0 F38.W0 9E /r	VFNMSUB132PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm0 and ymm2/mem, negate the multiplication result and subtract ymm1. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W0 AE /r	VFNMSUB213PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm0, negate the multiplication result and subtract ymm2/mem. Put the result in ymm0.
VEX.DDS.256.66.0 F38.W0 BE /r	VFNMSUB231PS ymm0, ymm1, ymm2/m256	Multiply packed single-precision floating-point values from ymm1 and ymm2/mem, negate the multiplication result and subtract ymm0. Put the result in ymm0.
VEX.DDS.128.66.0 F38.W1 9F /r	VFNMSUB132SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 AF /r	VFNMSUB213SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W1 BF /r	VFNMSUB231SD xmm0, xmm1, xmm2/m64	Multiply scalar double-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.

Opcode	Instruction	Description
VEX.DDS.128.66.0 F38.W0 9F /r	VFNMSUB132SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm0 and xmm2/mem, negate the multiplication result and subtract xmm1. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 AF /r	VFNMSUB213SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm0, negate the multiplication result and subtract xmm2/mem. Put the result in xmm0.
VEX.DDS.128.66.0 F38.W0 BF /r	VFNMSUB231SS xmm0, xmm1, xmm2/m32	Multiply scalar single-precision floating-point value in xmm1 and xmm2/mem, negate the multiplication result and subtract xmm0. Put the result in xmm0.

Table A-6. VEX-Encoded and Other General-Purpose Instruction Sets

Opcode	Instruction	Description
VEX.NDS.LZ.0F38. W0 F2 /r	ANDN r32a, r32b, r/m32	Bitwise AND of inverted r32b with r/m32, store result in r32a
VEX.NDS.LZ.0F38. W1 F2 /r	ANDN r64a, r64b, r/m64	Bitwise AND of inverted r64b with r/m64, store result in r64a
VEX.NDS.LZ.0F38. W0 F7 /r	BEXTR r32a, r/m32, r32b	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.NDS.LZ.0F38. W1 F7 /r	BEXTR r64a, r/m64, r64b	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a.
VEX.NDD.LZ.0F38. W0 F3 /3	BLSI r32, r/m32	Set all lower bits in r32 to "1" starting from bit 0 to lowest set bit in r/m32
VEX.NDD.LZ.0F38. W1 F3 /3	BLSI r64, r/m64	Set all lower bits in r64 to "1" starting from bit 0 to lowest set bit in r/m64
VEX.NDD.LZ.0F38. W0 F3 /2	BLSMSK r32, r/m32	Extract lowest bit from r/m32 and set that bit in r32
VEX.NDD.LZ.0F38. W1 F3 /2	BLSMSK r64, r/m64	Extract lowest bit from r/m64 and set that bit in r64
VEX.NDD.LZ.0F38. W0 F3 /1	BLSR r32, r/m32	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32
VEX.NDD.LZ.0F38. W1 F3 /1	BLSR r64, r/m64	Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64
VEX.NDS.LZ.0F38. W0 F5 /r	BZHI r32a, r/m32, r32b	Zero bits in r/m32 starting with the position in r32b, write result to r32a.

Opcode	Instruction	Description
VEX.NDS.LZ.0F38.W1 F5 /r	BZHI r64a, r/m64, r64b	Zero bits in r/m64 starting with the position in r64b, write result to r64a.
F3 0F BD /r	LZCNT r16, r/m16	Count the number of leading zero bits in r/m16, return result in r16
F3 0F BD /r	LZCNT r32, r/m32	Count the number of leading zero bits in r/m32, return result in r32
REX.W + F3 0F BD /r	LZCNT r64, r/m64	Count the number of leading zero bits in r/m64, return result in r64.
VEX.NDD.LZ.F2.0F 38.W0 F6 /r	MULX r32a, r32b, r/m32	Unsigned multiply of r/m32 with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F 38.W1 F6 /r	MULX r64a, r64b, r/m64	Unsigned multiply of r/m64 with RDX without affecting arithmetic flags.
VEX.NDS.LZ.F2.0F 38.W0 F5 /r	PDEP r32a, r32b, r/m32	Parallel deposit of bits from r32b using mask in r/m32, result is written to r32a.
VEX.NDS.LZ.F2.0F 38.W1 F5 /r	PDEP r64a, r64b, r/m64	Parallel deposit of bits from r64b using mask in r/m64, result is written to r64a
VEX.NDS.LZ.F3.0F 38.W0 F5 /r	PEXT r32a, r32b, r/m32	Parallel extract of bits from r32b using mask in r/m32, result is written to r32a.
VEX.NDS.LZ.F3.0F 38.W1 F5 /r	PEXT r64a, r64b, r/m64	Parallel extract of bits from r64b using mask in r/m64, result is written to r64a
VEX.LZ.0F3A.W0 F0 /r ib	RORX r32, r/m32, imm8	Rotate 32-bit r/m32 right imm8 times without affecting arithmetic flags.
VEX.LZ.0F3A.W1 F0 /r ib	RORX r64, r/m64, imm8	Rotate 64-bit r/m64 right imm8 times without affecting arithmetic flags.
VEX.NDS.LZ.F3.0F 38.W0 F7 /r	SARX r32a, r/m32, r32b	Shift r/m32 arithmetically right with count specified in r32b.
VEX.NDS.LZ.F3.0F 38.W1 F7 /r	SARX r64a, r/m64, r64b	Shift r/m64 arithmetically right with count specified in r64b.
VEX.NDS.LZ.66.0F 38.W0 F7 /r	SHLX r32a, r/m32, r32b	Shift r/m32 logically left with count specified in r32b.
VEX.NDS.LZ.66.0F 38.W1 F7 /r	SHLX r64a, r/m64, r64b	Shift r/m64 logically left with count specified in r64b.
VEX.NDS.LZ.F2.0F 38.W0 F7 /r	SHRX r32a, r/m32, r32b	Shift r/m32 logically right with count specified in r32b.
VEX.NDS.LZ.F2.0F 38.W1 F7 /r	SHRX r64a, r/m64, r64b	Shift r/m64 logically right with count specified in r64b.
F3 0F BC /r	TZCNT r16, r/m16	Count the number of trailing zero bits in r/m16, return result in r16
F3 0F BC /r	TZCNT r32, r/m32	Count the number of trailing zero bits in r/m32, return result in r32

Opcode	Instruction	Description
REX.W + F3 0F BC /r	TZCNT r64, r/m64	Count the number of trailing zero bits in r/m64, return result in r64.
66 0F 38 82 /r	INVPCID r32, m128	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 0F 38 82 /r	INVPCID r64, m128	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

Table A-7. New Instructions Introduced in Processors Code Named Ivy Bridge

Opcode	Instruction	Description
F3 0F AE /0	RDFSBASE r32	Read FS base register and place the 32-bit result in the destination register.
REX.W + F3 0F AE /0	RDFSBASE r64	Read FS base register and place the 64-bit result in the destination register.
F3 0F AE /1	RDGSBASE r32	Read GS base register and place the 32-bit result in destination register.
REX.W + F3 0F AE /1	RDGSBASE r64	Read GS base register and place the 64-bit result in destination register.
0F C7 /6	RDRAND r16	Read a 16-bit random number and store in the destination register.
0F C7 /6	RDRAND r32	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /6	RDRAND r64	Read a 64-bit random number and store in the destination register.
F3 0F AE /2	WRFSBASE r32	Write the 32-bit value in the source register to FS base register.
REX.W + F3 0F AE /2	WRFSBASE r64	Write the 64-bit value in the source register to FS base register.
F3 0F AE /3	WRGSBASE r32	Write the 32-bit value in the source register to GS base register.
REX.W + F3 0F AE /3	WRGSBASE r64	Write the 64-bit value in the source register to GS base register.
VEX.256.66.0F38.W0 13 /r	VCVTPH2PS ymm1, xmm2/m128	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.

Opcode	Instruction	Description
VEX.128.66.0F38. W0 13 /r	VCVTPH2PS xmm1, xmm2/m64	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.
VEX.256.66.0F3A. W0 1D /r ib	VCVTPS2PH xmm1/m128, ymm2, imm8	Convert eight packed single-precision floating-point value in ymm2 to packed half-precision (16-bit) floating-point value in xmm1/mem. Imm8 provides rounding controls.
VEX.128.66.0F3A. W0.1D /r ib	VCVTPS2PH xmm1/m64, xmm2, imm8	Convert four packed single-precision floating-point value in xmm2 to packed half-precision (16-bit) floating-point value in xmm1/mem. Imm8 provides rounding controls.

APPENDIX B
OPCODE MAP

Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table B-2 through Table B-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table B-7 through Table B-22.

NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

B.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table B-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table B-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table B-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section B.2.4, “Opcode Look-up Examples for One, Two, and Three-Byte Opcodes.”

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table B-2 and Table B-3, see Section B.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section B.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes

outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

B.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

B.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).

B The VEX.vvvv field of the VEX prefix selects a general purpose register.

C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F EFLAGS/RFLAGS Register.

G The reg field of the ModR/M byte selects a general register (for example, AX (000)).

H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.

I Immediate data: the operand value is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).

L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)

M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).

N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.

O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).

P The reg field of the ModR/M byte selects a packed quadword MMX technology register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).

S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).

U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).

Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

B.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).

b Byte, regardless of operand-size attribute.

c Byte or word, depending on operand-size attribute.

d Doubleword, regardless of operand-size attribute.

dq	Double-quadword, regardless of operand-size attribute.
p	32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd	128-bit or 256-bit packed double-precision floating-point data.
pi	Quadword MMX technology register (for example: mm0).
ps	128-bit or 256-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
qq	Quad-Quadword (256-bits), regardless of operand-size attribute.
s	6-byte or 10-byte pseudo-descriptor.
sd	Scalar element of a 128-bit double-precision floating data.
ss	Scalar element of a 128-bit single-precision floating data.
si	Doubleword integer register (for example: eax).
v	Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w	Word, regardless of operand-size attribute.
x	dq or qq based on the operand-size attribute.
y	Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z	Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

B.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

B.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

B.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table B-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section B.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section B.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Operand types are listed according to notations listed in Section B.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table B-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

Example B-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table B-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section B.4).

B.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table B-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table B-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table B-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section B.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section B.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section B.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table B-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

Example B-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H for a SHLD instruction using Table B-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).

- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

B.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table B-4 and Table B-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table B-4 or Table B-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table B-4 or Table B-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in B.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section B.2.

Example B-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table B-5.

- 66H is a prefix and 0F3AH indicate to use Table B-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  - Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

B.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a "v". If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix "v" is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript "v".

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.

B.2.5 Superscripts Utilized in Opcode Tables

Table B-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

Table B-1. Superscripts Utilized in Opcode Tables

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section B.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table B-6.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).

Table B-1. Superscripts Utilized in Opcode Tables

Superscript Symbol	Meaning of Symbol
v	VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

B.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table B-2 through Table B-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

Table B-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES^{i64}	POP ES^{i64}
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS^{i64}	POP SS^{i64}
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA^{i64}
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA^{i64}
4	INC^{i64} general register / REX^{o64} Prefixes eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH^{d64} general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1^{1A} Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	TEST Eb, Gb	Ev, Gv	XCHG Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	MOV immediate byte into byte register AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2^{1A} Eb, Ib	Ev, Ib	RETN^{f64} Iw	RETN^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} Gz, Mp VEX+1byte	Grp 11^{1A} - MOV Eb, Ib	Ev, Iz
D	Shift Grp 2^{1A} Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ Jb	IN AL, Ib	eAX, Ib	OUT Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3^{1A} Eb	Ev

Table A-2. One-byte Opcode Map: (08H — FFH) *

	8	9	A	B	C	D	E	F
0	OR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH CS[i64]	2-byte escape (Table A-3)
1	SBB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS[i64]	POP DS[i64]
2	SUB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=CS (Prefix)	DAS[i64]
3	CMP Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=DS (Prefix)	AAS[i64]
4	DEC[i64] general register / REX[o64] Prefixes eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB
5	POP[d64] into general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH[d64] Iz	IMUL Gv, Ev, Iz	PUSH[d64] Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc[f64], Jb- Short displacement jump on condition S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	MOV Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A[1A] POP[d64] Ev
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	CALLF[i64] Ap	FWAIT/ WAIT	PUSHF/D/Q[d64]/ Fv	POPF/D/Q[d64]/ Fv	SAHF	LAHF
A	TEST AL, Ib	rAX, Iz	STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Xv
B	MOV immediate word or double into word, double, or quad register rAX/r8, Iv	rCX/r9, Iv	rDX/r10, Iv	rBX/r11, Iv	rSP/r12, Iv	rBP/r13, Iv	rSI/r14, Iv	rDI/r15 , Iv
C	ENTER Iw, Ib	LEAVE[d64]	RETF Iw	RETF	INT 3	INT Ib	INTO[i64]	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL[f64] Jz	JMP near[f64] Jz	far[i64] Ap	short[f64] Jb	IN AL, DX	eAX, DX	OUT DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4[1A]	INC/DEC Grp 5[1A]

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) *

	pfx	0	1	2	3	4	5	6	7
0		Grp 6[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL[o64]	CLTS	SYSRET[o64]
1		vmovups	vmovups	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps[v1] Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps[v1] Mq, Vq
	66	vmovupd	vmovupd Wpd,Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx,Hx,Wx	vunpckhpd Vx,Hx,Wx	vmovhpd[v1] Vdq, Hq, Mq	vmovhpd[v1] Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2		MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3		WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4		CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy,Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12[1A])	(Grp 13[1A])	(Grp 14[1A])	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper[v] vzeroall[v]
	66	vpshufd Vx, Wx, Ib				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshufhw Vx, Wx, Ib							
	F2	vpshuflw Vx, Wx, Ib							

Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2[1B]		prefetchw(/1) Ev		
1		Prefetch[1C] (Grp 16[1A])							NOP Ev
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd,Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtsi2ss Vss, Hss, Ey		vcvttss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtsi2sd Vsd, Hsd, Ey		vcvttsd2si Gy, Wsd	vcvtsd2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvttps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtsd2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpcklqdq Vx, Hx, Wx	vpunpckhqdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsubpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx,Vx
	F3							vmovq Vq, Wq	vmovdqu Wx,Vx
	F2					vhaddps Vps, Hps, Wps	vhsubps Vps, Hps, Wps		

Table A-3. Two-byte Opcode Map: 80H — F7H (First Byte is 0FH) *

	pfx	0	1	2	3	4	5	6	7
8		Jcc[f64], Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH[d64] FS	POP[d64] FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG Eb, Gb	CMPXCHG Ev, Gv	LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb	MOVZX Gv, Ew
C		XADD Eb, Gb	XADD Ev, Gv	vcmpps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9[1A]
	66			vcmppd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib	
	F3			vcmpss Vss,Hss,Wss,Ib					
	F2			vcmpsd Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhuw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vx, Hx, Wx	vpsraw Vx, Hx, Wx	vpsrad Vx, Hx, Wx	vpavgw Vx, Hx, Wx	vpmulhuw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvttpd2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtdq2pd Vx, Wpd	
	F2							vcvtpd2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
8		Jcc[f64], Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH[d64] GS	POP[d64] GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15[1A])[1C]	IMUL Gv, Ev
B		*JMPE* (reserved for emulator on IPF)	Grp 10[1A] Invalid Opcode[1B]	Grp 8[1A] Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	MOVSX Gv, Ew
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
D		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmaxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpminsw Vx, Hx, Wx	vpor Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmaxsw Vx, Hx, Wx	vpxor Vx, Hx, Wx
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) *

	pfx	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phaddd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vx, Hx, Wx	vphaddw Vx, Hx, Wx	vphaddd Vx, Hx, Wx	vphaddsw Vx, Hx, Wx	vpmaddubsw Vx, Hx, Wx	vphsubw Vx, Hx, Wx	vphsubd Vx, Hx, Wx	vphsubsw Vx, Hx, Wx
1	66	pblendvb Vdq, Wdq			vcvtph2ps[V] Vx, Wx, Ib	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps[V] Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxwd Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
3	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxwd Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd[V] Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q[V] Vx, Hx, Wx	vpsravd[V] Vx, Hx, Wx	vpsllvd/q[V] Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
9	66	vgatherdd/q[V] Vx,Hx,Wx	vgatherqd/q[V] Vx,Hx,Wx	vgatherdps/d[V] Vx,Hx,Wx	vgatherqps/d[V] Vx,Hx,Wx			vfmaddsub132ps/d[V] Vx,Hx,Wx	vfmsubadd132ps/d[V] Vx,Hx,Wx
A	66							vfmaddsub213ps/d[V] Vx,Hx,Wx	vfmsubadd213ps/d[V] Vx,Hx,Wx
B	66							vfmaddsub231ps/d[V] Vx,Hx,Wx	vfmsubadd231ps/d[V] Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN[V] Gy, By, Ey	Grp 17[1A]		BZHI[V] Gy, Ey, By		BEXTR[V] Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX[V] Gy, Ey, By
	F3						PEXT[V] Gy, By, Ey	ADOX Gy, Ey	SARX[V] Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP[V] Gy, By, Ey	MULX[V] By,Gy,rDX,Ey	SHRX[V] Gy, Ey, By
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

Table A-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H) *

	pfx	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vx, Hx, Wx	vpsignw Vx, Hx, Wx	vpsignd Vx, Hx, Wx	vpmulhrsw Vx, Hx, Wx	vpermilps[V] Vx,Hx,Wx	vpermilpd[V] Vx,Hx,Wx	vtestps[V] Vx, Wx	vtestpd[V] Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss[V] Vx, Wd	vbroadcastsd[V] Vqq, Wq	vbroadcastf128[V] Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2	66	vpmuldq Vx, Hx, Wx	vpcmpeqq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps[V] Vx,Hx,Mx	vmaskmovpd[V] Vx,Hx,Mx	vmaskmovps[V] Mx,Hx,Vx	vmaskmovpd[V] Mx,Hx,Vx
3	66	vpminsb Vx, Hx, Wx	vpminsd Vx, Hx, Wx	vpminuw Vx, Hx, Wx	vpminud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
4									
5	66	vpbroadcastd[V] Vx, Wx	vpbroadcastq[V] Vx, Wx	vbroadcasti128[V] Vqq, Mdq					
6									
7	66	vpbroadcastb[V] Vx, Wx	vpbroadcastw[V] Vx, Wx						
8	66					vpmaskmovd/q[V] Vx,Hx,Mx		vpmaskmovd/q[V] Mx,Vx,Hx	
9	66	vfmadd132ps/d[V] Vx, Hx, Wx	vfmadd132ss/d[V] Vx, Hx, Wx	vfmsub132ps/d[V] Vx, Hx, Wx	vfmsub132ss/d[V] Vx, Hx, Wx	vfnmadd132ps/d[V] Vx, Hx, Wx	vfnmadd132ss/d[V] Vx, Hx, Wx	vfnmsub132ps/d[V] Vx, Hx, Wx	vfnmsub132ss/d[V] Vx, Hx, Wx
A	66	vfmadd213ps/d[V] Vx, Hx, Wx	vfmadd213ss/d[V] Vx, Hx, Wx	vfmsub213ps/d[V] Vx, Hx, Wx	vfmsub213ss/d[V] Vx, Hx, Wx	vfnmadd213ps/d[V] Vx, Hx, Wx	vfnmadd213ss/d[V] Vx, Hx, W	vfnmsub213ps/d[V] Vx, Hx, Wx	vfnmsub213ss/d[V] Vx, Hx, Wx
B	66	vfmadd231ps/d[V] Vx, Hx, Wx	vfmadd231ss/d[V] Vx, Hx, Wx	vfmsub231ps/d[V] Vx, Hx, Wx	vfmsub231ss/d[V] Vx, Hx, Wx	vfnmadd231ps/d[V] Vx, Hx, Wx	vfnmadd231ss/d[V] Vx, Hx, Wx	vfnmsub231ps/d[V] Vx, Hx, Wx	vfnmsub231ss/d[V] Vx, Hx, Wx
C									
D	66				VAESIMC Vdq, Wdq	VAESENC Vdq,Hdq,Wdq	VAESENCLAST Vdq,Hdq,Wdq	VAESDEC Vdq,Hdq,Wdq	VAESDECLAST Vdq,Hdq,Wdq
E									
F									
	66								
	F3								
	F2								
	66 & F2								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *

	pfx	0	1	2	3	4	5	6	7
0	66	vpermq^{V} Vqq, Wqq, Ib	vpermpd^{V} Vqq, Wqq, Ib	vpblendd^{V} Vx,Hx,Wx,Ib		vpermilps^{V} Vx, Wx, Ib	vpermilpd^{V} Vx, Wx, Ib	vperm2f128^{V} Vqq,Hqq,Wqq,Ib	
1	66					vpextrb Rd/Mb, Vdq, Ib	vpextrw Rd/Mw, Vdq, Ib	vpextrd/q Ey, Vdq, Ib	vextractps Ed, Vdq, Ib
2	66	vpinsrb Vdq,Hdq, Ry/Mb,Ib	vinsertps Vdq,Hdq, Udq/Md,Ib	vpinsrd/q Vdq,Hdq,Ey,Ib					
3									
4	66	vdpps Vx,Hx,Wx,Ib	vdppd Vdq,Hdq,Wdq,Ib	vmpsadbw Vx,Hx,Wx,Ib		vpclmulqdq Vdq,Hdq,Wdq,Ib		vperm2i128^{V} Vqq,Hqq,Wqq,Ib	
5									
6	66	vpcmpestrm Vdq, Wdq, Ib	vpcmpestri Vdq, Wdq, Ib	vpcmpistrm Vdq, Wdq, Ib	vpcmpistri Vdq, Wdq, Ib				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX^{V} Gy, Ey, Ib							

Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) *

	pfx	8	9	A	B	C	D	E	F
0									palignr Pq, Qq, Ib
	66	vroundps Vx,Wx,Ib	vroundpd Vx,Wx,Ib	vroundss Vss,Wss,Ib	vroundsd Vsd,Wsd,Ib	vblendps Vx,Hx,Wx,Ib	vblendpd Vx,Hx,Wx,Ib	vpblendw Vx,Hx,Wx,Ib	vpalignr Vx,Hx,Wx,Ib
1	66	vinsertf128[v] Vqq,Hqq,Wqq,Ib	vextractf128[v] Wdq,Vqq,Ib				vcvtps2ph[v] Wx, Vx, Ib		
2									
3	66	vinserti128[v] Vqq,Hqq,Wqq,Ib	vextracti128[v] Wdq,Vqq,Ib						
4	66			vblendvps[v] Vx,Hx,Wx,Lx	vblendvpd[v] Vx,Hx,Wx,Lx	vpblendvb[v] Vx,Hx,Wx,Lx			
5									
6									
7									
8									
9									
A									
B									
C									
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure B-1) as an extension of the opcode.

mod	nnn	R/M

Figure B-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table B-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

B.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example B-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table B-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example B-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table B-2, Table B-3 and Table B-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table B-3) reveals that this opcode is in Group 7 of Table B-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table B-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

B.4.2 Opcode Extension Tables

See Table B-6 below.

Table B-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B		POP							
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B		TEST Ib/Iz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	CALLN[f64] Ev	CALLF Ep	JMPN[f64] Ev	JMPF Mp	PUSH[d64] Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110)					SWAPGS[o64] (000) RDTSCP (001)
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	VMPTRST Mq
		11B								RDRAND Rv	RDSEED Rv
0F B9	10	mem									
		11B									
C6	11	mem		MOV Eb, Ib							
		11B									XABORT (000) Ib
C7		mem		MOV Ev, Iz							
		11B									XBEGIN (000) Jz

Table B-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F 71	12	mem									
		11B				psrlw Nq, Ib		psraw Nq, Ib		psllw Nq, Ib	
			66			vpsrlw Hx,Ux,Ib		vpsraw Hx,Ux,Ib		vpsllw Hx,Ux,Ib	
0F 72	13	mem									
		11B				psrld Nq, Ib		psrad Nq, Ib		pslld Nq, Ib	
			66			vpsrld Hx,Ux,Ib		vpsrad Hx,Ux,Ib		vpslld Hx,Ux,Ib	
0F 73	14	mem									
		11B				psrlq Nq, Ib				psllq Nq, Ib	
			66			vpsrlq Hx,Ux,Ib	vpsrldq Hx,Ux,Ib			vpsllq Hx,Ux,Ib	vpslldq Hx,Ux,Ib
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11B							lfence	mfence	sfence
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry				
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B									
VEX.0F38 F3	17	mem			BLSR^{V} By, Ey	BLSMSK^{V} By, Ey	BLSI^{V} By, Ey				
		11B									

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table B-7 through Table B-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1-and 2-byte opcodes (see B.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

B.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example B-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table B-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example B-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section B.4.
- In Table B-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

B.5.2 Escape Opcode Instruction Tables

Tables are listed below.

B.5.2.1 Escape Opcodes with D8 as First Byte

Table B-7 and B-8 contain maps for the escape instruction opcodes that begin with D8H. Table B-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure B.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.2 Escape Opcodes with D9 as First Byte

Table B-9 and B-10 contain maps for escape instruction opcodes that begin with D9H. Table B-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

.

Table B-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	FXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.3 Escape Opcodes with DA as First Byte

Table B-11 and B-12 contain maps for escape instruction opcodes that begin with DAH. Table B-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-11 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.4 Escape Opcodes with DB as First Byte

Table B-13 and B-14 contain maps for escape instruction opcodes that begin with DBH. Table B-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.5 Escape Opcodes with DC as First Byte

Table B-15 and B-16 contain maps for escape instruction opcodes that begin with DCH. Table B-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure B-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.6 Escape Opcodes with DD as First Byte

Table B-17 and B-18 contain maps for escape instruction opcodes that begin with DDH. Table B-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.7 Escape Opcodes with DE as First Byte

Table B-19 and B-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table B-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMPP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0).	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

B.5.2.8 Escape Opcodes with DF As First Byte

Table B-21 and B-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table B-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure B-1) selects the instruction.

Table B-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table B-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table B-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

INDEX

B

BEXTR - Bit Field Extract 7-4
BLSI - Extract Lowest Set Isolated Bit 7-6
BLSMSK - Get Mask Up to Lowest Set Bit 7-8
BLSR - Reset Lowest Set Bit 7-10
Brand information 2-59
  processor brand index 2-63
  processor brand string 2-60
BZHI - Zero High Bits Starting with Specified Bit Position 7-12

C

Cache and TLB information 2-54
Cache Inclusiveness 2-37
CLFLUSH instruction
  CPUID flag 2-53
CMOVcc flag 2-53
CMOVcc instructions
  CPUID flag 2-53
CMPXCHG16B instruction
  CPUID bit 2-49
CMPXCHG8B instruction
  CPUID flag 2-52
CPUID instruction 2-34, 2-53
  36-bit page size extension 2-53
  APIC on-chip 2-52
  basic CPUID information 2-35
  cache and TLB characteristics 2-35, 2-54
  CLFLUSH flag 2-53
  CLFLUSH instruction cache line size 2-47
  CMPXCHG16B flag 2-49
  CMPXCHG8B flag 2-52
  CPL qualified debug store 2-49
  debug extensions, CR4.DE 2-52
  debug store supported 2-53
  deterministic cache parameters leaf 2-36, 2-39, 2-40, 2-41, 2-42
  extended function information 2-42
  feature information 2-51
  FPU on-chip 2-52
  FSAVE flag 2-53
  FXRSTOR flag 2-53
  HT technology flag 2-54
  IA-32e mode available 2-43
  input limits for EAX 2-44
  L1 Context ID 2-49
  local APIC physical ID 2-47
  machine check architecture 2-53
  machine check exception 2-52
  memory type range registers 2-52
  MONITOR feature information 2-58
  MONITOR/MWAIT flag 2-49
  MONITOR/MWAIT leaf 2-37, 2-38, 2-39, 2-40
  MWAIT feature information 2-58
  page attribute table 2-53
  page size extension 2-52
  performance monitoring features 2-59

physical address bits . . . 2-44
physical address extension . . . 2-52
power management . . . 2-58, 2-59
processor brand index . . . 2-47, 2-60
processor brand string . . . 2-43, 2-60
processor serial number . . . 2-53
processor type field . . . 2-46
PTE global bit . . . 2-53
RDMSR flag . . . 2-52
returned in EBX . . . 2-47
returned in ECX & EDX . . . 2-47
self snoop . . . 2-54
SpeedStep technology . . . 2-49
SS2 extensions flag . . . 2-54
SSE extensions flag . . . 2-54
SSE3 extensions flag . . . 2-49
SSSE3 extensions flag . . . 2-49
SYSENTER flag . . . 2-52
SYSEXIT flag . . . 2-52
thermal management . . . 2-58, 2-59
thermal monitor . . . 2-49, 2-53, 2-54
time stamp counter . . . 2-52
using CPUID . . . 2-34
vendor ID string . . . 2-44
version information . . . 2-35, 2-58
virtual 8086 Mode flag . . . 2-52
virtual address bits . . . 2-44
WRMSR flag . . . 2-52

F

Feature information, processor . . . 2-34
FMA operation . . . 2-7, 2-8
FXRSTOR instruction
CPUID flag . . . 2-53
FXSAVE instruction
CPUID flag . . . 2-53

H

Hardware Lock Elision (HLE) . . . 8-1, 8-2
Hyper-Threading Technology
CPUID flag . . . 2-54

I

IA-32e mode
CPUID flag . . . 2-43
Intel Transactional Synchronization . . . 8-1
INVPCID - Invalidate Processor Context ID . . . 7-29

L

L1 Context ID . . . 2-49
LZCNT - Count the Number of Leading Zero Bits . . . 7-14

M

Machine check architecture
CPUID flag . . . 2-53

description 2-53
MMX instructions
CPUID flag for technology 2-53
Model & family information 2-58
MONITOR instruction
CPUID flag 2-49
feature data 2-58
MOVNTDQA - Load Double Quadword Non-Temporal Aligned Hint 5-221
MPSADBW - Multiple Sum of Absolute Differences 5-8
MULX - Unsigned Multiply Without Affecting Flags 7-16
MWAIT instruction
CPUID flag 2-49
feature data 2-58

O

Opcodes
addressing method codes for B-2
extensions B-20
extensions tables B-21
group numbers B-20
integers
one-byte opcodes B-10
two-byte opcodes B-12
key to abbreviations B-2
look-up examples B-5, B-20, B-23
ModR/M byte B-20
one-byte opcodes B-5, B-10
opcode maps B-1
operand type codes for B-3
register codes for B-4
superscripts in tables B-8
two-byte opcodes B-6, B-7, B-12
x87 ESC instruction opcodes B-23

P

PABSB/PABSW/PABSD - Packed Absolute Value 5-17
PACKSSWB/PACKSSDW - Pack with Signed Saturation 5-21
PACKUSDW - Pack with Unsigned Saturation 5-26
PACKUSWB - Pack with Unsigned Saturation 5-30
PADDB/PADDW/PADDD/PADDQ - Add Packed Integers 5-34
PADDSB/PADDSW - Add Packed Signed Integers with Signed Saturation 5-39
PADDUSB/PADDUSW - Add Packed Unsigned Integers with Unsigned Saturation 5-42
PALIGNR - Byte Align 5-45
PAND - Logical AND 5-48
PANDN - Logical AND NOT 5-50
PAVGB/PAVGW - Average Packed Integers 5-52
PBLENDVB - Variable Blend Packed Bytes 5-55
PBLENDW - Blend Packed Words 5-60
PCMPEQB/PCMPEQW/PCMPEQD/PCMPEQQ - Compare Packed Integers for Equality 5-63
PCMPGTB/PCMPGTW/PCMPGTD/PCMPGTQ - Compare Packed Integers for Greater Than . . 5-68
PDEP - Parallel Bits Deposit 7-18
Pending break enable 2-54
Performance-monitoring counters
CPUID inquiry for 2-59
PEXT - Parallel Bits Extract 7-20
PHADDSW - Packed Horizontal Add with Saturation 5-77
PHADDW/PHADDD - Packed Horizontal Add 5-73
PHSUBSW - Packed Horizontal Subtract with Saturation 5-84
PHSUBW/PHSUBD - Packed Horizontal Subtract 5-80
PMADDUBSW - Multiply and Add Packed Integers 5-87

PMADDWD - Multiply and Add Packed Integers . . . 5-89
PMAXSB/PMAXSW/PMAXSD - Maximum of Packed Signed Integers . . . 5-92
PMAXUB/PMAXUW/PMAXUD - Maximum of Packed Unsigned Integers . . . 5-97
PMINSB/PMINSW/PMINSD - Minimum of Packed Signed Integers . . . 5-102
PMINUB/PMINUW/PMINUD - Minimum of Packed Unsigned Integers . . . 5-107
PMOVMSKB - Move Byte Mask . . . 5-112
PMOVSX - Packed Move with Sign Extend . . . 5-114
PMOVZX - Packed Move with Zero Extend . . . 5-120
PMULDQ - Multiply Packed Doubleword Integers . . . 5-126
PMULHRSW - Multiply Packed Unsigned Integers with Round and Scale . . . 5-129
PMULHUW - Multiply Packed Unsigned Integers and Store High Result . . . 5-133
PMULHW - Multiply Packed Integers and Store High Result . . . 5-136
PMULLW/PMULLD - Multiply Packed Integers and Store Low Result . . . 5-139
PMULUDQ - Multiply Packed Unsigned Doubleword Integers . . . 5-144
POR - Bitwise Logical Or . . . 5-147
PSADBW - Compute Sum of Absolute Differences . . . 5-149
PSHUFB - Packed Shuffle Bytes . . . 5-152
PSHUFD - Shuffle Packed Doublewords . . . 5-155
PSHUFHW - Shuffle Packed High Words . . . 5-158
PSHUFLW - Shuffle Packed Low Words . . . 5-161
PSIGNB/PSIGNW/PSIGND - Packed SIGN . . . 5-164
PSLLDQ - Byte Shift Left . . . 5-170
PSLLW/PSLLD/PSLLQ - Bit Shift Left . . . 5-172
PSRAW/PSRAD - Bit Shift Arithmetic Right . . . 5-179
PSRLDQ - Byte Shift Right . . . 5-184
PSRLW/PSRLD/PSRLQ - Shift Packed Data Right Logical . . . 5-186
PSUBB/PSUBW/PSUBD/PSUBQ - Packed Integer Subtract . . . 5-193
PSUBSB/PSUBSW - Subtract Packed Signed Integers with Signed Saturation . . . 5-199
PSUBUSB/PSUBUSW - Subtract Packed Unsigned Integers with Unsigned Saturation . . . 5-202
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ - Unpack High Data . . . 5-205
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ - Unpack Low Data . . . 5-212
PXOR - Exclusive Or . . . 5-219

R

RDMSR instruction
  CPUID flag . . . 2-52
RORX - Rotate Right Logical Without Affecting Flags . . . 7-22

S

SARX/SHLX/SHRX - Shift Without Affecting Flags . . . 7-24
Self Snoop . . . 2-54
SIB byte
  32-bit addressing forms of . . . 4-12
SpeedStep technology . . . 2-49
SSE extensions
  CPUID flag . . . 2-54
SSE2 extensions
  CPUID flag . . . 2-54
SSE3
  CPUID flag . . . 2-49
SSE3 extensions
  CPUID flag . . . 2-49
SSSE3 extensions
  CPUID flag . . . 2-49
Stepping information . . . 2-58
SYSENTER instruction
  CPUID flag . . . 2-52
SYSEXIT instruction
  CPUID flag . . . 2-52

T

Thermal Monitor
CPUID flag . . . 2-54
Thermal Monitor 2 . . . 2-49
CPUID flag . . . 2-49
Time Stamp Counter . . . 2-52
TZCNT - Count the Number of Trailing Zero Bits . . . 7-27

V

VBROADCAST - Broadcast Floating-Point Data . . . 5-224
VBROADCASTF128/I128 - Broadcast 128-Bit Data . . . 5-226
Version information, processor . . . 2-34
VEX . . . 5-2
VEXTRACTI128 - Extract packed Integer Values . . . 5-243
VEX.B . . . 5-2
VEX.L . . . 5-3
VEX.mmmmm . . . 5-2
VEX.pp . . . 5-3
VEX.R . . . 5-4
VEX.vvvv . . . 5-3
VEX.W . . . 5-2
VEX.X . . . 5-2
VFMADD132PD/VFMADD213PD/VFMADD231PD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values . . . 6-2
VFMADD132SD/VFMADD213SD/VFMADD231SD - Fused Multiply-Add of Scalar Double-Precision Floating-Point Values . . . 6-10
VFMADD132SS/VFMADD213SS/VFMADD231SS - Fused Multiply-Add of Scalar Single-Precision Floating-Point Values . . . 6-13
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD - Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values . . . 6-16
VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS - Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values . . . 6-20
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD - Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values . . . 6-32
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS - Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values . . . 6-36
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD - Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values . . . 6-40
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS - Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values . . . 6-43
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD - Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values . . . 6-24
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD - Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values . . . 6-46
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS - Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values . . . 6-50
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD - Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values . . . 6-54
VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD - Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values . . . 6-60
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD - Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values . . . 6-68
VGATHERDPD/VGATHERQPD - Gather Packed DP FP Values Using Signed Dword/Qword Indices 5-259
VGATHERDPS/VGATHERQPS - Gather Packed SP FP values Using Signed Dword/Qword Indices 5-265
VINSERTF128- Insert Packed Integer Values . . . 5-245
VPBLENDD - Blend Packed Dwords . . . 5-228
VPBROADCAST - Broadcast Integer Data . . . 5-230
VPERM2I128 - Permute Integer Values . . . 5-241
VPERMD - Full Doublewords Element Permutation . . . 5-235
VPERMPD - Permute Double-Precision Floating-Point Elements . . . 5-237

VPERMPS - Permute Single-Precision Floating-Point Elements. 5-238
VPERMQ - Qwords Element Permutation. 5-240
VPGATHERDD/VPGATHERQD - Gather Packed Dword Values Using Signed Dword/Qword Indices 5-271
VPGATHERDQ/VPGATHERQQ - Gather Packed Qword values Using Signed Dword/Qword Indices 5-277
VPMASKMOV - Conditional SIMD Integer Packed Loads and Stores 5-247
VPSLLVD/VPSLLVQ - Variable Bit Shift Left Logical 5-251
VPSRAVD - Variable Bit Shift Right Arithmetic. 5-254
VPSRLVD/VPSRLVQ - Variable Bit Shift Right Logical. 5-256

W

WBINVD/INVD bit. 2-37
WRMSR instruction
CPUID flag 2-52

X

x87 FPU
instruction opcodes. B-23
XABORT - Transaction Abort 8-16
XACQUIRE/XRELEASE - Hardware Lock Elision Prefix Hints 8-12
XBEGIN - Transaction Begin 8-18
XEND - Transaction End 8-22
XFEATURE_ENALBED_MASK 2-2
XRSTOR 1-2, 2-2, 2-59, 3-1, 5-7
XSAVE. 1-2, 2-2, 2-3, 2-4, 2-5, 2-6, 2-12, 2-50, 2-59, 3-1, 5-7
XTEST - Test If In Transactional Execution 8-24